FIFTH SEMIANNUAL TECHNICAL REPORT

(15 December 1971 - 15 June 1972)

FOR THE PROJECT

"RESEARCH IN

STORE AND FORWARD COMPUTER NETWORKS"

Principal Investigator

and Project Manager:

HOWARD FRANK (516) 671-9580

ARPA Order No. 1523

Contractor: Network Analysis Corporation

Contract No. DAHC 15-70-C-0120

Effective Date: 15 October 1969

Expiration Date: 15 October 1972

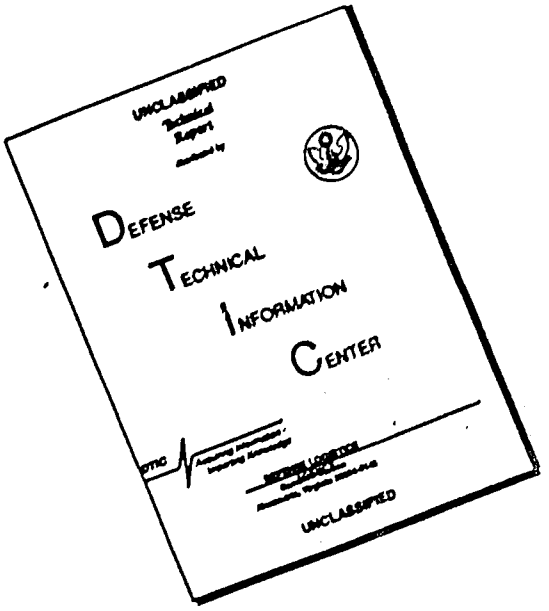
Sponsored by

Advanced Research Projects Agency

Department of Defnese

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
Network Analysis Corporation
Beechwood, Old Tappan Rd., Glen Cove, NY 11542

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP
None

3. REPORT TITLE
Semiannual Repct No. 5
June 1972

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Semiannual Technical Report, 15 Dec. 1971 - 15 June 1972

5. AUTHOR(S) (Last name, first name, initial)
Network Analysis Corporation

6. REPORT DATE
June 1972

7a. TOTAL NO. OF PAGES
91

7b. NO. OF REFS
27

8a. CONTRACT OR GRANT NO.
DAHC-15-70-C-0120

b. PROJECT NO.
ARPA Order No. 1523

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
ARPA SEMIANNUAL REPORT NO. 5

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
----

10. AVAILABILITY/LIMITATION NOTICES
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency,
Department of Defense

13. ABSTRACT This report studies reliability properties of store-and-forward networks, analysis of network reliability and algorithms for minimum spanning trees. A study of the tradeoffs between network size, connectivity, and component reliability shows that large networks reliability will be a major, and perhaps dominant, design problem. Recursive analysis techniques for loop and tree combinations greatly reduce analysis cost, while improved methods for generating minimum spanning trees have a similar effect for this fundamental network problem.

14. KEY WORDS
Computer networks, throughput, cost, reliability, survivability, ARPA Computer Network, store-and-forward

Security Classification

TABLE OF CONTENTS

SUMMARY

Technical Problem

The Network Analysis Corporation contract with the Advanced Research Projects Agency incorporates the following objectives: To determine the most economical configurations for the ARPANET, to study the properties of store and forward networks and to develop procedures for analysis and design of reliable computer communication networks.

General Methodology

The heart of the research program has been a dual attack on basic network theoretical problems and the development of computational techniques for the study of large networks.

Technical Results

Some of the results accomplished during the reporting period are:

- A study of the tradeoffs between network size, network connectivity and component reliability was completed. This study indicates that reliability will be a major and perhaps dominant issue for large network design.

- A new method for reliability analysis which uses a recursive technique has been developed to handle a large class of networks composed of loops and trees. This method allows a wide variety of reliability criteria to be evaluated simultaneously at a small fraction of the cost of previously known methods.
- New and improved computational techniques for finding "minimum spanning trees", (a fundamental network problem) were derived. This computation is a basic ingredient in many large scale network algorithms.

Department of Defense Implications

Communication networks for meeting Department of Defense requirements involve huge network structures that present techniques are inadequate to handle. The results of the reporting period highlight the role that reliability will play in such networks, provide new techniques for the analysis of large Defense Department networks and meet some of the computational requirements for large scale network design.

Implications for Further Research

This report shows that for very large networks, cost/reliability considerations must be given equal importance to cost/throughput considerations. This means that there will be

a need to develop dramatically different network design procedures to insure availability of resources in a large network. The requirements of the new procedures, while not yet well defined, indicate that computation breakthroughs for a number of basic network problems will be necessary.

I. RELIABILITY AND LARGE COMPUTER NETWORKS

1. Introduction and Summary

The major considerations in the system design of a computer network such as the ARPANET are:

1) Cost

2) Throughput

3) Delay and response time

4) Network reliability

While it is essential to consider each of these constraints, it often results that several are automatically satisfied for designs satisfying the remaining. Initially, this was the case for the ARPANET. The delay and response time was adequately considered by slightly derating the line capacities of the 50 kilobit links and the reliability was adequate if there were at least two node disjoint paths between each pair of nodes. Thus, the cost-throughput tradeoff was the overriding consideration. Given these conditions, it is possible to design very efficient networks in a reasonable amount of computing time. However, it is becoming evident that as the ARPANET increases in size, the reliability constraints are beginning to limit design choices. It may even become that the cost-reliability tradeoff may replace the cost-throughput

tradeoff as the basic design consideration. While for small versions of the ARPANET, any design with at least two node disjoint paths between each node pair and sufficient throughput would necessarily be reliable enough, initial investigations indicate that for large networks sufficient reliability automatically implies sufficient throughput. In any case, it is clear that reliability constraints will play an ever increasing role in the design process as the ARPANET becomes larger. Considering this, it is quite sobering to note that many large communication networks are being designed with little consideration of network reliability (as distinguished from component or element reliability).

Reliability analysis of computer networks is concerned with the dependence of the reliability of the network on the reliability of its nodes and links. Element reliability is easily defined as, for example, the fraction of time the element is operable, or as by the mean time between failures and expected repair time. The proper measure of network reliability is not as clear and simple. Several possible measures are: the number of elements which must be removed to disconnect the network, the probability that the network will be disconnected, the expected fraction of node pairs which can communicate through the network, and the expected

throughput of the network subject to element failures. The above measures are listed in order of their computational complexity. Many other measures can and have been suggested. A whole other class of measures arise when the nodes are not of equal importance. as in centralized networks or hierarchal networks. In a centralized network, one may be interested in the expected number of nodes which can communicate with a central node. More general criteria arise when different node pairs are weighted by their importance. For example, communication between ILLIAC IV and certain other nodes will be of high priority in the ARPANET. Most of our analysis will deal with exepcted fraction of node pairs communicating although in many cases any of the other criteria mentioned could be used.

Node failures can affect network reliability in two ways. First, if a node fails, clearly it cannot communicate with any other node in the network. Thus, if there are NN nodes in the network and one fails, a minimum of NN-1 node pairs cannot communicate independent of the network structure. In the next section we establish a simple formula for measuring this effect. Changing the network configuration has no effect on this component of network reliability. Another effect of node failures is that the failed nodes destroy some potential communication paths between other pairs of nodes. Link failures also affect network reliability in the second way.

In the next section, we survey the reliability situation for small versions of the ARPANET. In Section 3 we enumerate several independent pieces of evidence which point out the increasing role of reliability considerations in larger ARPANETS. In the final section of the chapter, the implications of this trend are discussed.

2. Reliability of Small to Medium Networks (NN≤50)

The initial design procedure for the ARPANET controlled reliability by insisting that there be at least two node disjoint paths between every pair of nodes. Later computations proved that this implied almost perfect reliability in the following sense. Suppose node i in the network is inoperative a fraction p_i of the time for i=1,..., NN. Then a lower bound for the expected number of node pairs which cannot communicate is equal to the expected number of node pairs not communicating in a complete network where each node pair is joined by an invulnerable link. No addition or redistribution of links can reduce the expected number of node pairs not communicating below this value. For small nets, the existence of two node disjoint paths between each pair of nodes invariably resulted in an expected number of node pairs not communicating very near the lower bound. Thus, the addition of more links for reliability purposes was not justified. The calculation of this important lower bound is as follows:

Let each node i of a network with NN nodes have a probability p_i of failing. Then, the expected number of node pairs in which one or both nodes have failed is

$$\sum \{1-(1-p_i)(1-p_j)\} .$$

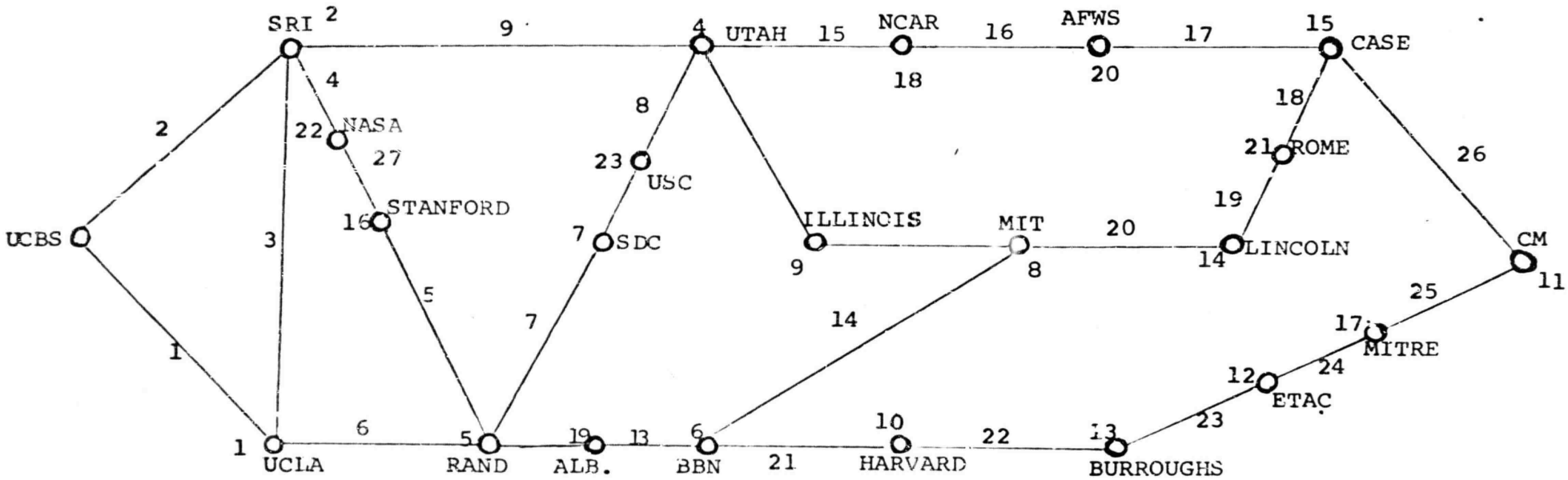
If $p_i = p$ for $i=1,\ldots, NN$, $i<j$ then the expected number is

$$NN(NN-1)\{1-(1-p)^2\} = NN(NN-1)[2p(1-p)]$$

and the expected fraction of node pairs with at least one node failed is $[2p(1-p)]$. Two important implications of this simple result deserve to be emphasized. First, the expected fraction of non-communicating node pairs cannot be reduced below $[2p(1-p)]$, and second this lower bound is invariant with respect to the size of the network.

To fix these ideas and to give specific examples of the reliability characteristics of small nets, we consider two versions of the ARPANET. The first is a 23 node network that has been thoroughly analyzed as a common measuring point or standard for the various reliability analysis techniques. The second network is a medium size network of 33 nodes in which for the first time an additional link was considered mainly for reliability reasons. The 23 node network is represented in Figure 1.1. This design had a yearly line cost of $847,000 for its 28 lines and a throughput of 9.9 Kbits/node

23 Node Network



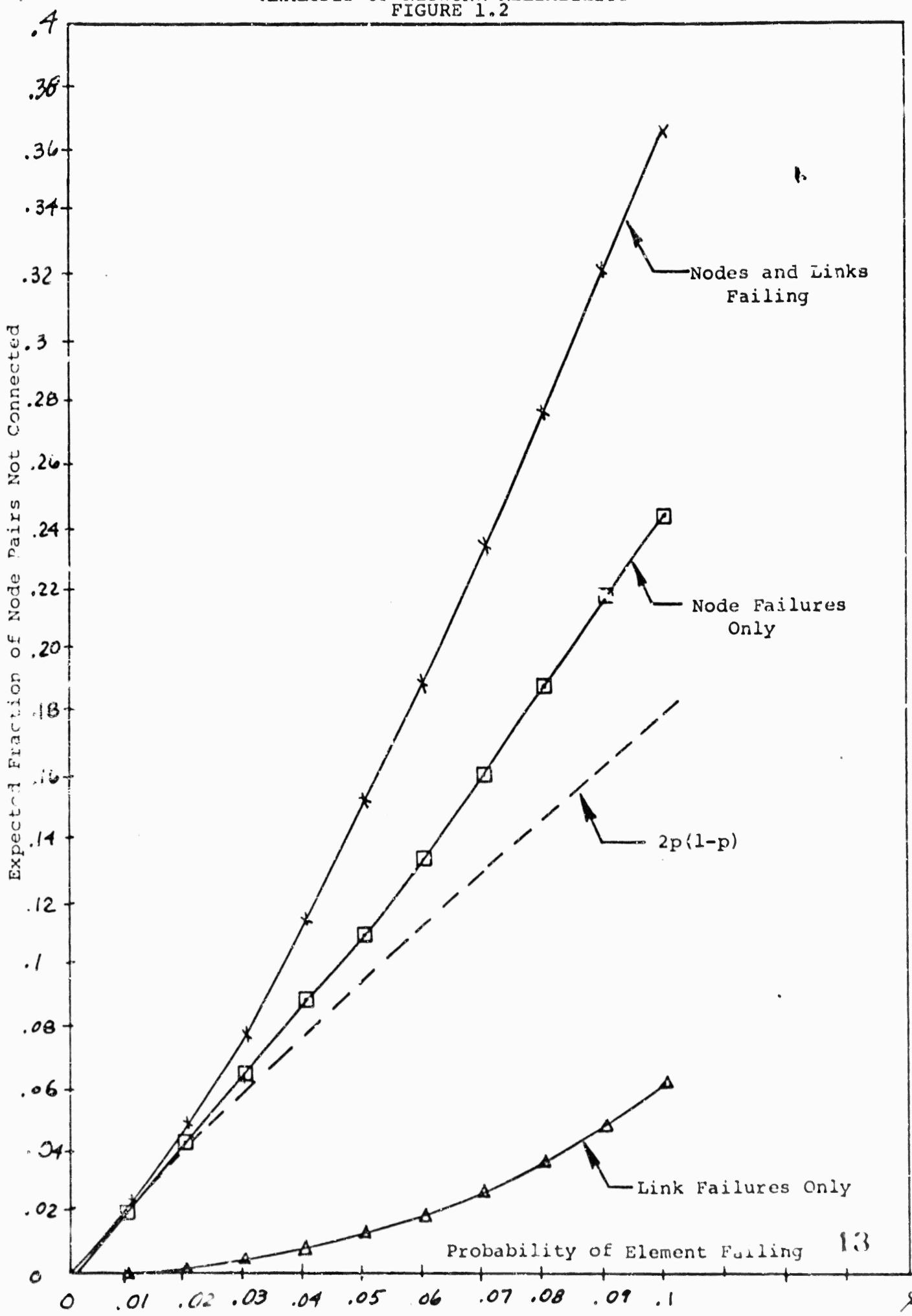
Base Network for Reliability Analysis

FIGURE 1.1

assuming uniform traffic between nodes. We will assume a base element failure probability 0.02 which is a close approximation to currently measured values. Then, 2p(1-p) equals 0.0396 for p=.02 and hence the expected fraction of node pairs not communicating must be at least (.0396)(23)(22)/2 equals 10.0188. In Figure 1.2 the expected fraction of node pairs not communicating as a function of element failure probability is shown. Also shown is the expected fraction of node pairs not communicating when only links fail, when only nodes fail and finally when the curve 2p(1-p) is plotted. For p = .02, the expected fraction of node pairs not communicating is 0.049.

In the case where only nodes fail the expected fraction is .0427 and for only links failing .0018. Remembering that 2p(1-p) = .0396, we see that 80% of the node pairs which cannot communicate can be ascribed to purely the fact that one of the nodes of the pair in question has failed. Thus, the improvement in reliability to be gained by changing the network configuration is minor. Nevertheless, several strategies for improving reliability were examined. The most vulnerable section of the 23 node network is the long string of nodes from node 6 (BBN) to node 15 (CASE) along the bottom of Figure 1.1. The first idea was to add a link from node 13

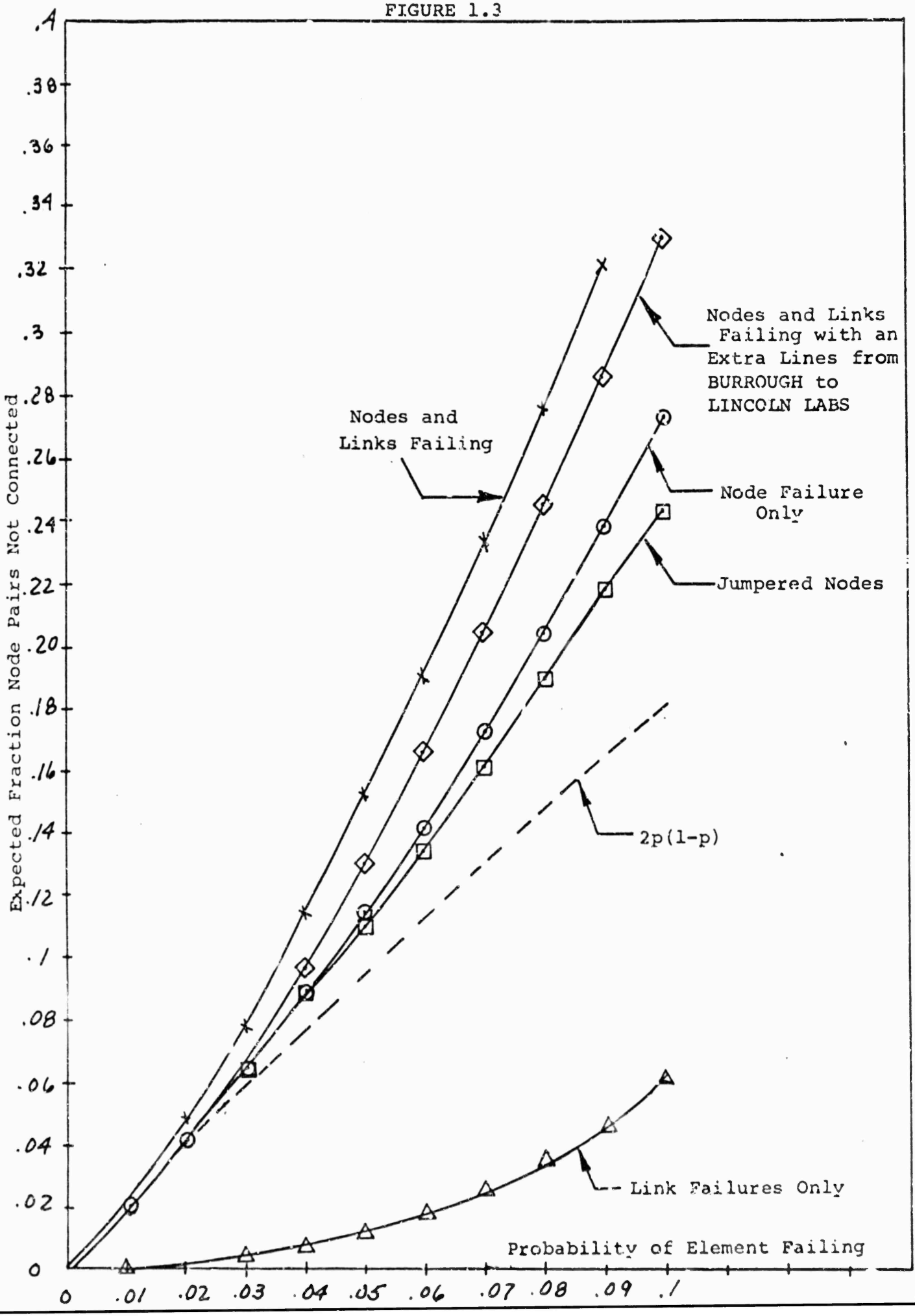
ANALYSIS OF NETWORK RELIABILITY
FIGURE 1.2



(BURROUGHTS) to node 14 (LINCOLN). The second idea was to install hardware at the IMPs so that if an IMP failed, traffic could be routed around it in one direction connecting two of the incident links. Any remaining links are effectively blocked. The results of these analyses are shown in Figure 1.3. For p=.02 the improvement is negligible and does not justify the cost of implementation although for higher values of p the improvement becomes more significant. The expected fraction of non-communicating node pairs is a purely topological reliability measure since it does not completely reflect the degradation of throughput due to element failures. The most detailed level of analysis of reliability incorporates element failures, flow requirements, routing, acceptable delays and other pertinent network characteristics. In order to test the adequacy of the ARPANET under the most stringent of conditions, a reliability analysis treating these factors was performed. The effect on throughput at average delay of 0.2 seconds was examined by removing nodes and links from the network and applying the NAC routing and analysis algorithms to the remaining network. The nominal throughput of the 23 node network with all elements operable is 11.5 KBPS/node. When nodes and links are failing with p=.02, the expected

ANALYSIS OF NETWORK RELIABILITY

FIGURE 1.3



11.

throughput is at least 9.0 KBPS/node. These results again show that for small networks, reliability is not a dominant factor.

Figures 1.4 and 1.5 depict a 33 node network. For the network shown in Figure 1.4 the difference between the expected fraction of node pairs not communicating = .058 and $2p(1-p)$ = .040 is almost double the difference for the 23 node network so that improving the reliability by changing the network configuration becomes marginally feasible. An extra link from FT.BEL to ABER increased the cost by a little over 1% and increased the reliability by almost 10%. The resulting network is shown in Figure 1.5. Thus, even for a network with only 33 nodes, it is becoming necessary to consider reliability in more detail than the "two connectivity" criteria. For $p > .02$ it is even more important.

3. Reliability Trends for Large Networks

While for smaller networks and low element failure probabilities ($p \leq .02$), it was found that designing the network with at least two node disjoint paths between each node pair for throughput in the range 8-15 kilobits/second/node guaranteed sufficient reliability; as networks become larger this simple approach fails. The first experiments which indicated this

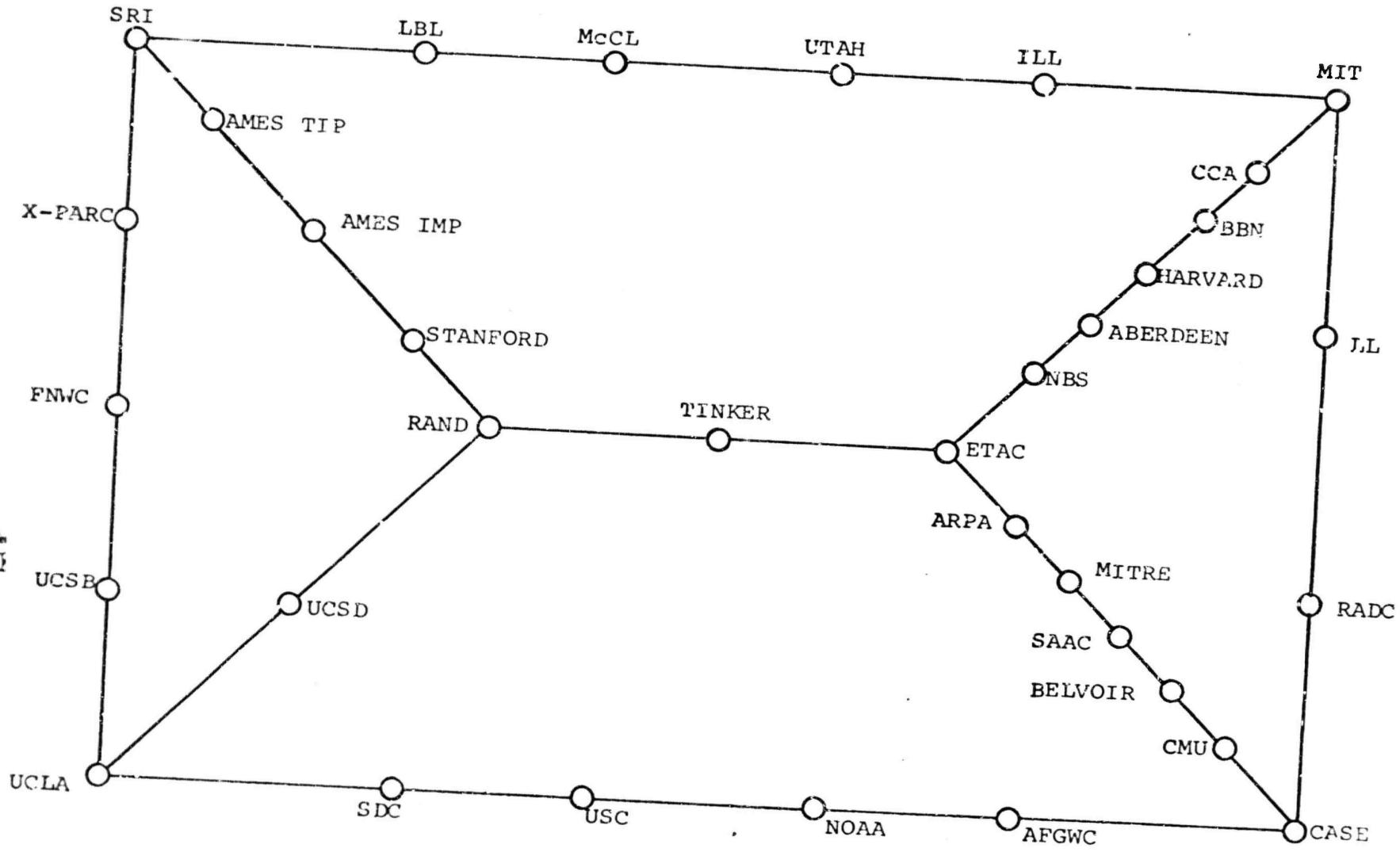

SRI
LBL
McCL
UTAH
ILL
MIT
AMES TIP
X-PARC
AMES IMP
CCA
BBN
HARVARD
ABERDEEN
LL
STANFORD
NBS
FNWC
RAND
TINKER
ETAC
ARPA
MITRE
UCSB
UCSD
RADC
SAAC
BELVOIR
CMU
UCLA
SDC
USC
NOAA
AFGWC
CASE

FIGURE 1.4

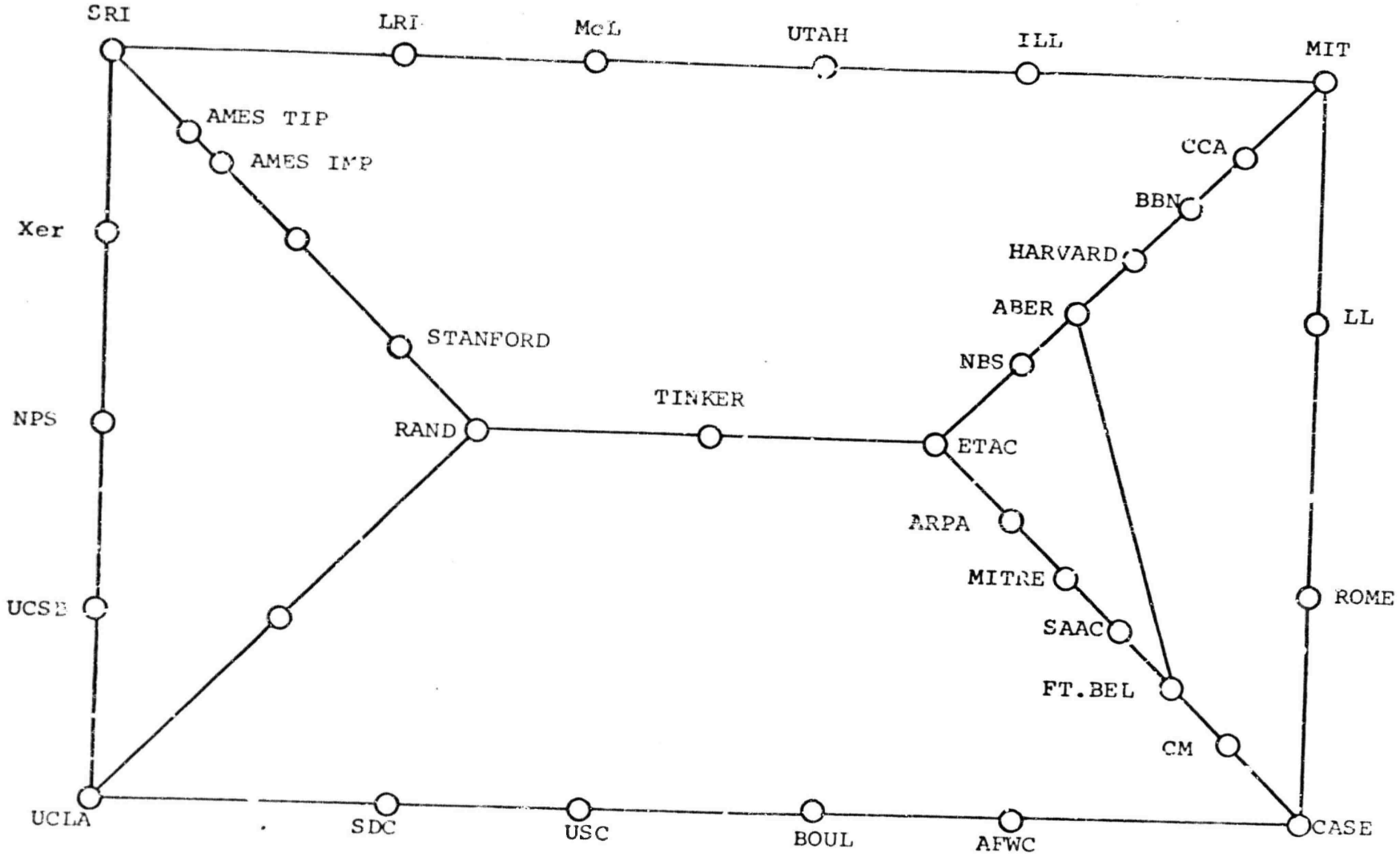

SRI
LRI
McL
UTAH
ILL
MIT
AMES TIP
AMES IMP
CCA
BBN
Xer
HARVARD
ABER
LL
STANFORD
NBS
TINKER
NPS
RAND
ETAC
ARPA
MITRE
ROME
UCSB
SAAC
FT.BEL
CM
UCLA
SDC
USC
BOUL
AFWC
CASE

FIGURE 1.5

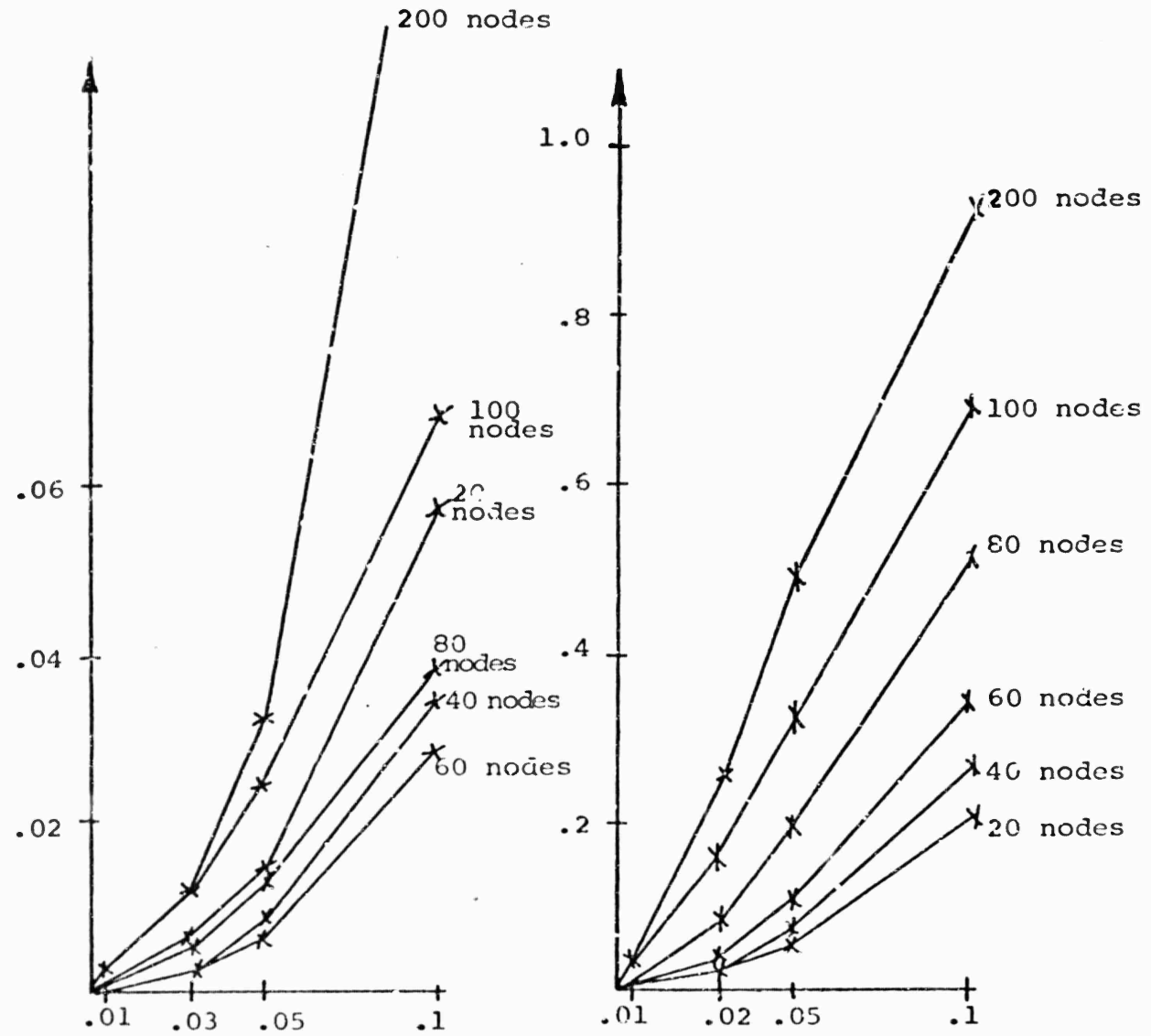
started with low cost networks of 20,40,60,80,100 and 200 nodes with throughput approximately 8 KBPS/node designed by NAC's network design program with the reliability constraint of two node disjoint paths. The results are shown in Figure 1.6 when nodes are perfectly reliable. As measured by the fraction of node pairs not communicating, the reliability actually increased with the number of nodes up to 60 nodes at which point the reliability began to decrease. As is evident, the decrease in reliability is dramatic even though nodes have been assumed to be perfect.

Figures 1.7 through 1.10 show the results of analysis of a family of two and three connected networks containing from 20 nodes to 200 nodes. The networks analyzed contain 20, 40, 60, 80, 100 and 200 nodes. However, a continuous line is drawn for visual convenience. On the curve in Figure 1.8 for p=0.2, the simulation/analysis error is indicated by vertical bars with length equal to 4 times the standard deviation. If the simulation results were normally distributed, this would corresponde to a 95% confidence interval. It can be seen from Figures 1.7 to 1.10 that when there are 3 node disjoint paths between every pair of nodes, the unreliability is close to the ideal minimum which results from only the node failures within the sampling error except for p = .1 where the 3 node disjoint paths curve is just beginning to depart from the idea curve. From these,

FIGURE 1.6
NETWORK RELIABILITY AS A FUNCTION OF NUMBER OF NODES

Fraction of node pairs not communicating versus probability of link failure

Probability of Network being disconnected versus probability of Link Failing



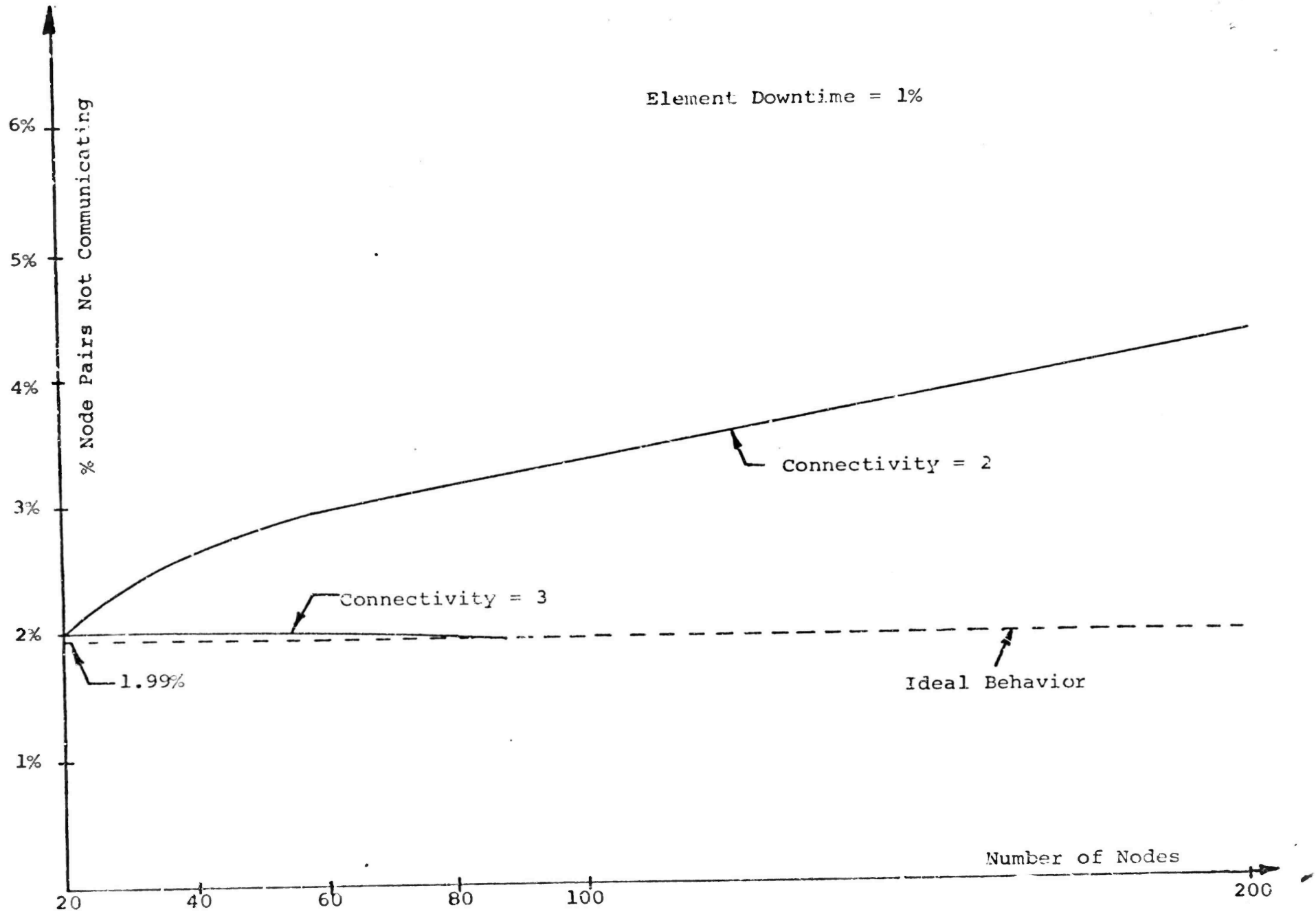

Element Downtime = 1%
% Node Pairs Not Communicating
6%
5%
4%
3%
2%
1%
Connectivity = 2
Connectivity = 3
1.99%
Ideal Behavior
Number of Nodes
20
40
60
80
100
200

FIGURE 1.7

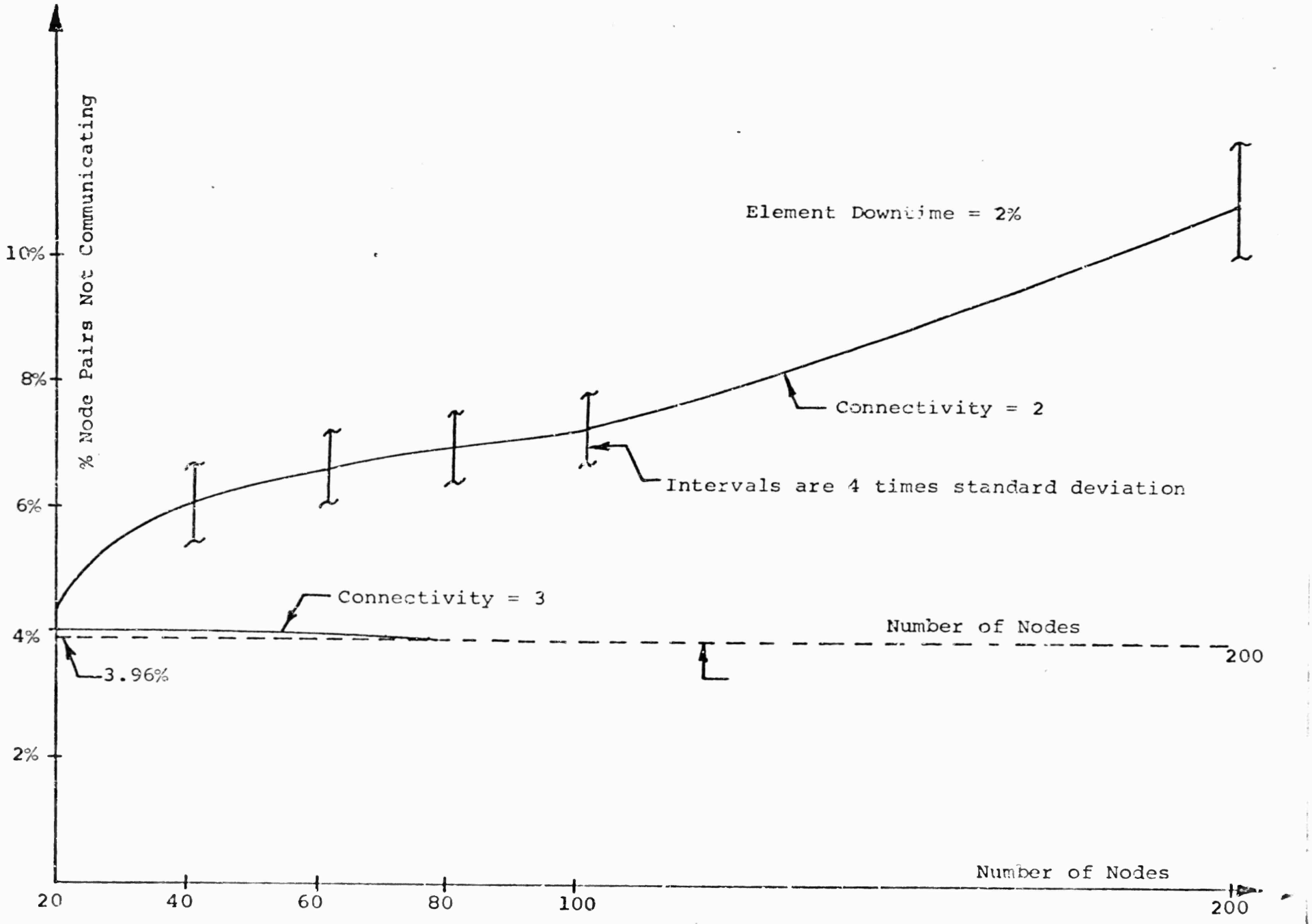

% Node Pairs Not Communicating
10%
8%
6%
4%
2%
Element Downtime = 2%
Connectivity = 2
Intervals are 4 times standard deviation
Connectivity = 3
Number of Nodes
200
3.96%
Number of Nodes
20
40
60
80
100
200

FIGURE 1.8

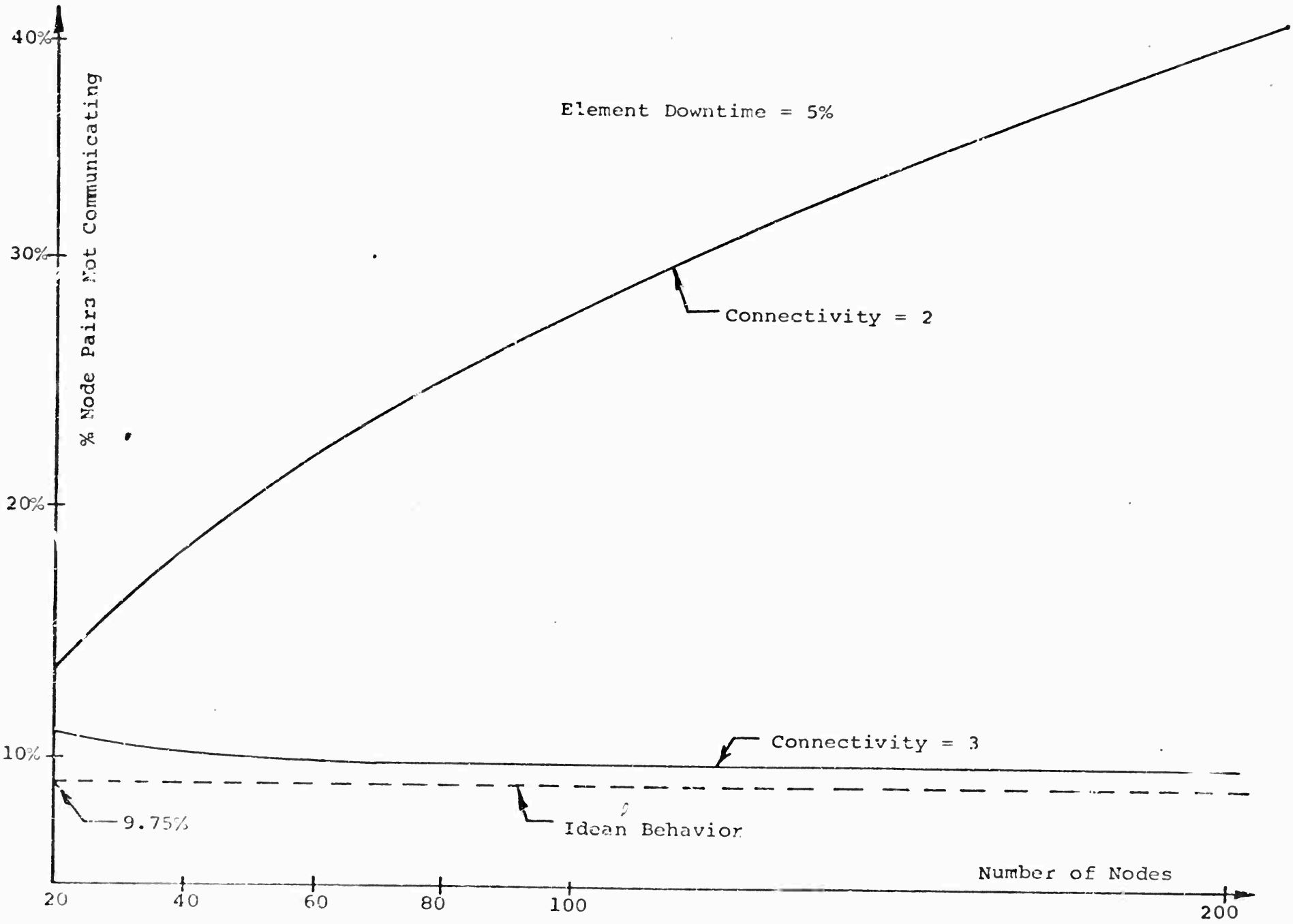

40%
30%
20%
10%
% Node Pairs Not Communicating
Element Downtime = 5%
Connectivity = 2
Connectivity = 3
9.75%
Idean Behavior
Number of Nodes
20
40
60
80
100
200

FIGURE 1.9

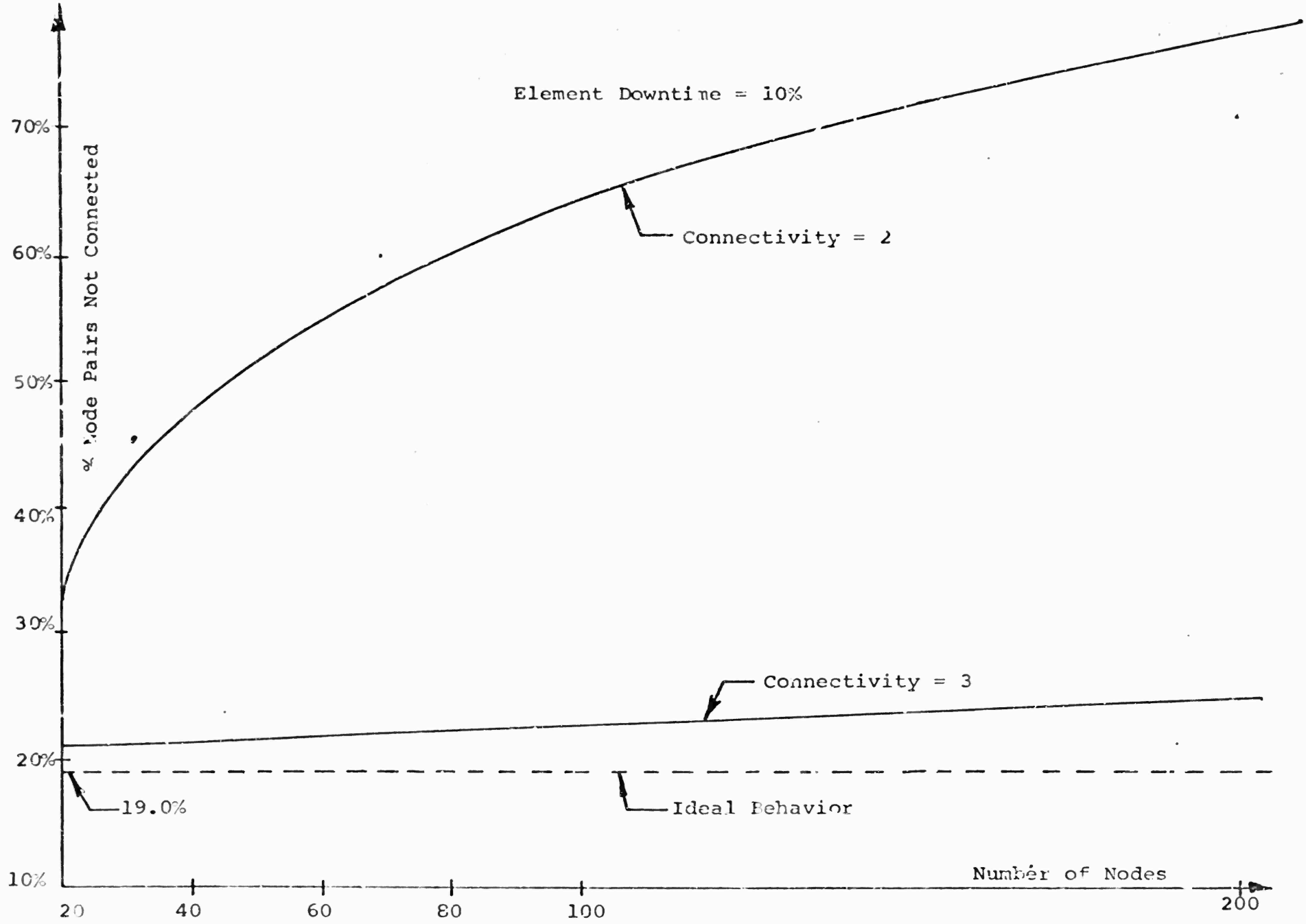

Element Downtime = 10%
% Node Pairs Not Connected
70%
60%
50%
40%
30%
20%
10%
Connectivity = 2
Connectivity = 3
19.0%
Ideal Behavior
Number of Nodes
20
40
60
80
100
200

FIGURE 1.10

we can conclude that requiring 3 node disjoint paths between every pair of nodes is sufficient to essentially guarantee an optimal reliability with respect to link allocations for networks with less than 200 nodes and for element failure probabilities of less than 0.1. Whether the use of this criterion would result in expensive over-design should be further investigated. In many cases, it is clear that this could occur so it is worthwhile to develop rapid reliability analysis methods which can be carried out repeatedly in the design process. Unfortunately, at present as fast as the current reliability analysis techniques have become, it is still infeasible to employ them in an iterative design process.

4. Implications for Further Research

Fast effective methods have been developed for analyzing the reliability of networks [ARPA Semi-Annual Reports 2, 3 and 4]. Recently, as will be described in the next chapter even more efficient analysis techniques have been developed. While these methods are effective for quite large networks, they are still too slow for use in an iterative design procedure. Recursive methods suitable for networks composed

from loops and trees are orders of magnitude faster and offer hope for use in design. These new methods are described in the next chapter. These recursive methods can be used in a hybrid manner with simulation using decomposition techniques. Networks which can be analyzed by recursion can also be used as control variates in simulation of general networks. Research is progressing in these areas.

The selective "hardening" of important nodes in a computer network is being studied quantitatively. It is clear that the only way to decrease the 2p(1-p) lower bound on the fraction of non-communicating node pairs is to increase the reliability of the nodes themselves. One way of doing this is to put a backup IMP at each node. Since this is usually prohibitively expensive, one can select a subset of nodes where backup can be provided on the basis of a reliability-cost tradeoff.

If for very large networks the cost-reliability tradeoff is the dominant factor in network design, replacing the cost-throughput tradeoff, there will obviously need to be dramatic changes in network design procedures. The surface has been barely broken in this area.

II. RECURSIVE ANALYSIS OF NETWORK RELIABILITY

1. Introduction and Summary

The network structure of many common communication networks can be represented as a composite of simple loops and trees. Reliability analysis of such networks can be carried out very quickly and efficiently by a new recursion approach described in this chapter. Moreover, a wide variety of reliability measures can be obtained using the same general method. The measures studied here are:

(i) the expected number of nodes communicating with a central node called a "root",

(ii) the expected number of node pairs communicating,

(iii) the expected number of node pairs communicating by a path through the central node,

(iv) the probability that operating nodes can communicate through the root,

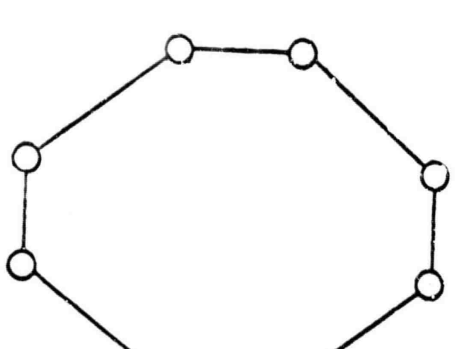
(v) the probability that operating nodes are connected.

Many other measures are possible.

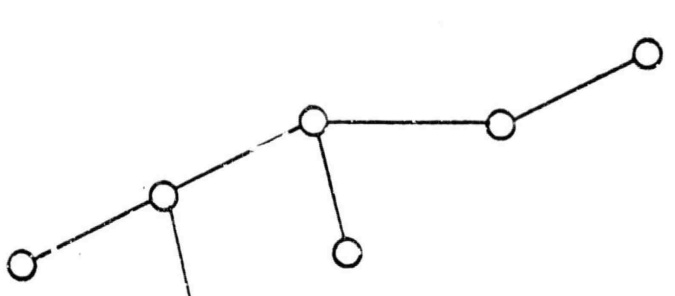
In Figure 2.1 some of the many network structures that can be analyzed using recursion are illustrated. In addition, even if a network does not have this precise structure, the

FIGURE 2.1

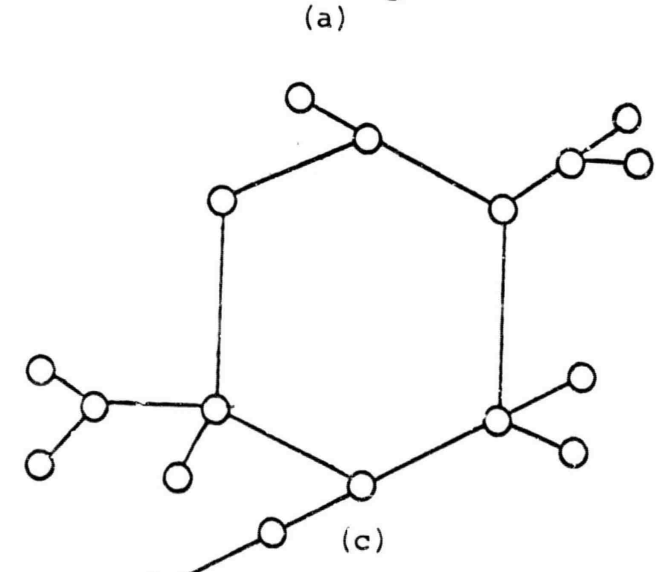
COMPOSITE LOOP AND TREE STRUCTURES



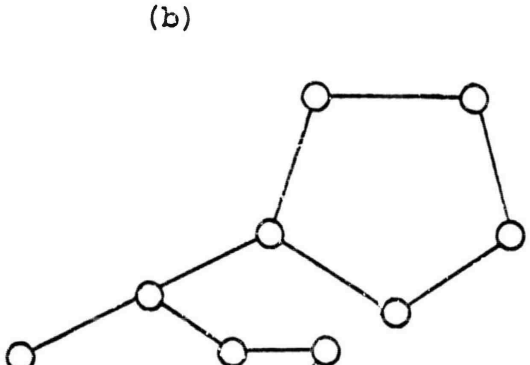
(a)



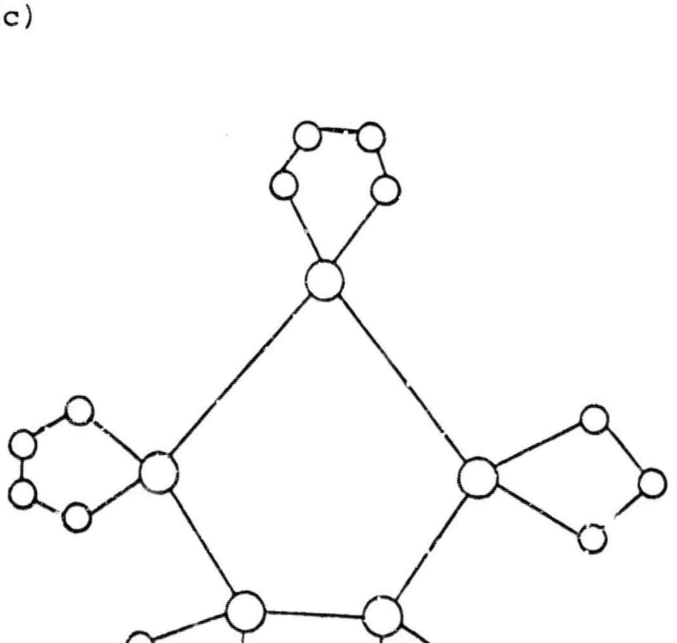
(b)



(c)



(d)



(e)

reliability of the network can often be approximated by the reliability of such a network or a hybrid computation using recursion on the tree and loop parts of the network together with simulation for the other parts can be carried out. (This generalized approach is now under study). These techniques then offer a very powerful tool in the analysis of network reliability.

2. Terminology

We will develop a very general class of recursive methods for a wide variety of reliability criteria. To do this it is very economical to employ a recursive characterization of rooted trees [Knuth:1968, Section 2.3].[1]

Definition: A rooted tree is a finite set T of one or more nodes such that:

(a) There is one specially designated node called the root of the tree, root (T); and

(b) The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets T_1, T_2, T_3, ..., T_m, and each of these sets in turn is a rooted tree. The trees T_1, ..., T_m are called subtrees of the root.

---

[1] The terminology of Knuth is somewhat different from ours.

As Knuth points out there are several models other than the obvious one, a tree graph with a distinguished node, but we will confine ourselves to tree graphs. To make this association more explicit we introduce some more terminology. The root of a tree, J, is said to be the father of the root of each of the subtrees of J. The root, I, of a subtree of J is said to be a son of J. Figure 2.2 depicts such a rooted tree graph where links are shown between fathers and their sons. A link is a pair of nodes one of which is the father of the other. Thus node 1 is the root of the entire tree. Node 2 is the root of the only subtree of 1 and hence 2 is the son of 1 and 1 is the father of 2. The corresponding subtree of 1 is determined by the nodes $\{2,3,4,5,6,7,8,9,10\}$. Node 2 has two subtrees on $\{3,4,5\}$ and $\{6,7,8,9,10\}$ with roots 3 and 6 respectively. Node 3 has two subtrees $\{4\}$ and $\{5\}$. Node 4 has no subtrees.

Since we will be dealing with computer methods of solution, it is necessary to impose a linear ordering for storage purposes. This will be done by a father function. Suppose we have a network on NN nodes, $\{1,2,\ldots,NN\}$, and for each node I except 1 we have a node F(I), the father of I, such that F(I)< I and (I,F(I)) is a link in the network. Then F defines NA=NN-1 links and in

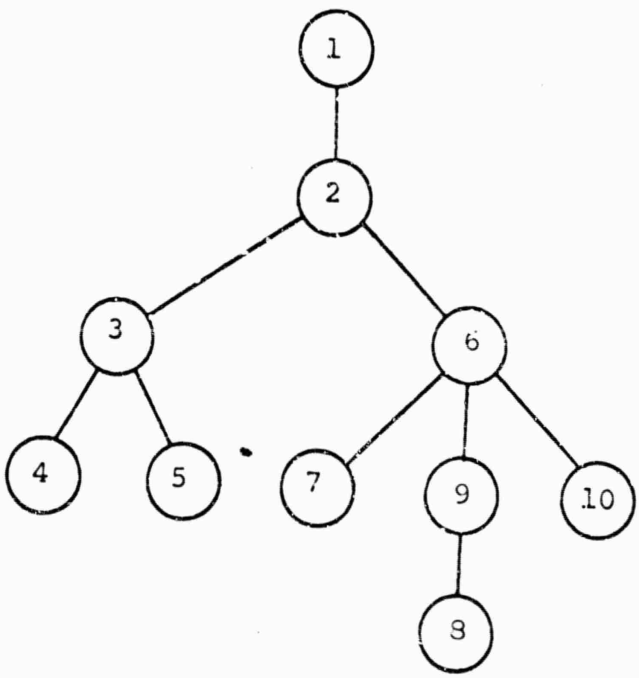


FIGURE 2.2

fact, the existence of a father function F is a necessary and sufficient condition for the network to be a rooted tree. The special node 1 (which has no father) is of course the root of the tree (sometimes called the patriarch). Associated with each node I is a rooted subtree consisting of nodes with greater numbers which are connected to I by a path passing through nodes with labels $\geq$ I. In Table 2.1 the father function for the tree in Figure 2.2 is given.

3. Recursive Computations on Trees

We now want to calculate the reliability of a tree network assuming the reliability of its elements, nodes and links, are known. It is not immediately obvious what the "reliability of a tree" should mean; we will consider several meanings. However, the general approach in each case will be the same. Considering the tree to be a rooted tree in the sense of Knuth, we associate a state vector with the root of each of the subtrees. We then define a set of recursion relations which yield the state vector of a rooted tree given the state of its subtrees. For subtrees consisting of single nodes the state is obvious. We then join the rooted subtrees into larger and larger rooted subtrees using the recursion relations until the state of the entire network is obtained.

I	F(I)
1	- -
2	1
3	2
4	3
5	3
6	2
7	6
8	7
9	6
10	6

Father Function

TABLE 2.1

Deriving the recurrence relations is somewhat mechanical also. It comes simply from considering the situation depicted in Figure 2.3. We have two subtrees one with root I and the other having as its root J=F(I). We assume the state of I and J are known and we wish to compute the state of J relative to the tree obtained by joining I and J by the link (I,J).

To illustrate the technique let us consider the first and easiest criterion. Namely, we wish to know the expected number of nodes which can communicate with the root node 1. We assume we have associated with each node I a probability of node failure PN(I) and a probability QN(I)=1-PN(I) of the node being present. Similarly, for the link (I,F(I)) we have probabilities PL(I) and QL(I) of the link failing and being operative respectively. The state vector of a subtree with root I is, in this case, a scalar, S(I) which is the expected number of nodes in the subtree which communicate with the root I, including I. To derive the recurrence relations we consider two subtrees with I and J=F(I) as roots, respectively. We then want to derive the state of the new subtree obtained by joining I and J together by (I,J). Let S(I) and S(J) be the known states for the two subtrees and S(J)' the resulting state. If the link (I,J) and the node J are operational S(J)'=S(I)+S(J); if not then

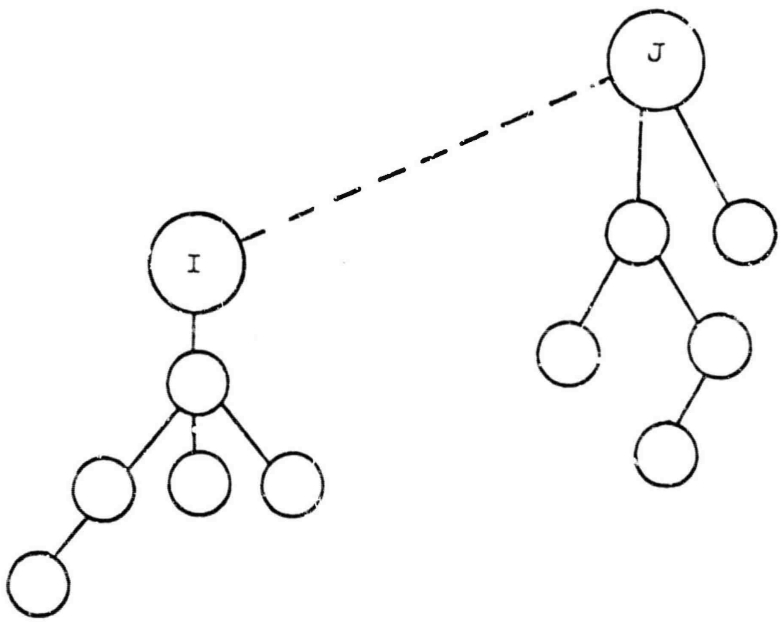


FIGURE 2.3

S(J)'=S(J). Putting the two together we have the recurrence relation: S(J)' =S(J)+S(I)QN(J)QL(I) where QN(J) is the probability that node J is operative and QL(I) is the probability that the link (I,J) is operative. Now all that remains is to put this in the form of an algorithm:

Step 0: (Initialization) Set S(I)=QN(I) (the probability that node I is working), I=1,..., NN. Set I=NN. Go to Step 1.

Step 1: Let J=F(I), and set S(J) to S(J)+S(I)QN(J)QL(I); go to Step 2.

Step 2: Set I to I-1. If I=1, stop; otherwise, go to Step 1.

When the algorithm stops S(1) is the expected number of nodes communicating with node 1 (counting node 1).

For our next criterion we compute the expected number of node pairs communicating. For this criterion we utilize a two dimensional state vector. We will use, as before, S(I) to be the expected number of nodes in the subtree which communicate with I, and a new state component T(I) which is the expected number of node pairs communicating in the subtree. The recursion relation for S(J) is as before S(J)'=S(J)+S(I)QN(J)QL(I). The recursion relation for T(J) is T(J)'=T(I)+T(J)+S(I)S(J)QL(I) since we have the same pairs communicating as before and if the link (I,J) is operating S(I) nodes in one tree can communicate with S(J) nodes of the other for S(I)S(J) additional node pairs.

The resulting algorithm is:

Step 0: (Initialization) Set S(I)=QN(I), T(I)=0, I=1,..., NN. Set I=NN. Go to Step 1.

Step 1: Let J=F(I); set T(J) to T(I)+T(J)+S(I)S(J)QL(I), and then set S(J) to S(J)+S(I)QN(J)QL(I). Go to Step 2.

Step 2: Set I to I-1. If I=1, stop; otherwise, go to Step 1.

T(1) ends up with the desired result. Note in Step 1, T(J) must be updated before S(J).

In many real systems node pairs can communicate only through the root. So for our next criterion we consider the expected number of node pairs which are connected by a path through the root. To analyze this case we consider a state component R(I) in place of T(I), where R(I) is the expected number of node pairs (pairs including I are allowed) both of which are connected to the root node I. S(I) has the same meaning as before. The recurrence relation for S(I) also remains unchanged. The recurrence relation for R(I) is R(J)'=R(J)+(S(I)S(J)+R(I)QN(J))QL(I). The algorithm needs only to be modified by changing the recurrence relation for T(J) in Step 1 to the one for R(J). The state components for this last criterion are illuminating. For if one knows the number of nodes connected to the root, say n, then the number of node pairs communicating through the root is

n(n-1)/2. This would seem to imply that either S(I) or R(I) could be eliminated and a state vector with one component would be possible. This is not the case because the expectation operation does not commute with squaring; that is, Exp[n(n-1)/2] $\neq$ (Exp n) (Exp n -1)/2, in general, for n random.

We now turn to a class of reliability criteria related to whether the network is connected or not. The first result is immediate: the probability QC of the tree being connected is

$$(1)\quad QC = \prod_{1}^{NN} QN(I) \prod_{2}^{NN} QL(I).$$

If we don't insist that the entire network be connected but only the subnetwork involving operative nodes be connected we get a new probability $\widetilde{QC}$. The calculation is more interesting in this case. Here we need a state vector for each subtree with 3 components. They are:

N(I) - The probability that all nodes in the subtree are failed.

C(I) - The probability that the (non-null) set of operative nodes, including the root of the subtree, are connected.

B(I) - The probability that the root of the subtree is failed and the set (non-null) of operative nodes in the subtree is connected.

N(I), C(I), and B(I) account for all tree networks whose operative nodes communicate.

The recurrence relations in this case are:

(2a) $C(J)'=C(I)C(J)QL(I)+C(J)N(I)$

(2b) $N(J)'=N(I)N(J)$

(2c) $B(J)'=B(J)N(I)+B(I)N(J)+C(I)N(J)$

As we mentioned before, often in practical situations all communication has to take place through the root node. So another interesting reliability condition is the probability, QR, that all operating nodes can communicate with the root. As can be seen from the definition of C, $QR=C(1)+N(1)$.

An algorithm for obtaining both criteria is:

Step 0: (Initialization) Set $N(I)=PN(I)$, $C(I)=QN(I)$, $B(I)=0$, $I=1, \ldots, NN$. Set $I=NN$. Go to Step 1.

Step 1: Let $J=F(I)$. Using equations (2), recalculate $B(J)$, $C(J)$, and $N(J)$, in that order. (Note that the order of calculations is important as calculations should be done with the old values of $B(J)$, $C(J)$, and $N(J)$.) Go to Step 2.

Step 2: Set $I=I-1$. If $I=1$, stop; otherwise, go to Step 1.

After the algorithm terminates, we obtain the probability of all operating nodes communicating by $\widetilde{QC}=C(1)+B(1)+N(1)$ and the probability of all operating nodes communicating with the root by $QR=C(1)+N(1)$.

We summarize the various algorithms in Table 2.2. The algorithms for finding the reliability measures discussed in this section were coded in FORTRAN IV and executed on a CDC-6600. The average running time for a 500 node tree was 1.5 seconds.

4. Trees with Weighted Nodes

In the previous section it was assumed that the nodes in the tree were all equal. In many cases it is desirable to assign a weight, W(I), to each node, I. As an example instead of wishing to calculate the expected number of nodes communicating with the root suppose we desired the expected amount of traffic which could reach the root where each node, I, generates W(I) units of traffic. To calculate this, the state variable is S, just as before, the only difference being that the initial conditions S(I)=W(I)QN(I) replaces the old initial conditions S(I)=QN(I). (W(I) could also represent the number of terminals at node I.)

It is possible, by the use of a weighting function, to extend the algorithms of the previous section to include the case where the "nodes" of the tree themselves represent trees, or indeed, more highly connected graphs. In this case, in Step 0, we initialize the state vector of each "node" of the network to the value of the state vector of the subnetwork we

TABLE 2.2

SUMMARY OF RELIABILITY MEASURES

Measure	Given By	State Vector
Exp(number of nodes communicating with root)	S(1)	S
Exp(number of node pairs communicating)	T(1)	S,T
Exp(number of node pairs communicating through root)	R(1)	S,R
Prob(operating nodes can communicate through root)	N(1)+C(1)	N,C
Prob(operating nodes are connected)	N(1)+C(1)+B(1)	N,C,B

Component	Recursion Relation	Initial Condition
S	S(J)'=S(J)+S(I)QN(J)QL(I)	S(I)=QN(I)
T	T(J)'=T(J)+T(I)+S(I)S(J)QL(I)	T(I)=0
R	R(J)'=R(J)+(R(I)QN(J)+S(I)S(J))QL(I)	R(I)=0
N	N(J)'=N(I)N(J)	N(I)=PN(I)
C	C(J)'=(C(I)QL(I)+N(I))C(J)	C(I)=QN(I)
B	B(J)'=B(J)N(I)+(BI)+C(I))N(J)	B(I)=0

are treating as a"node." Thus, in the previous example, we initialize the value of S(I) to the expected number of nodes communicating with node I in the subnetwork we are treating as a "node". In general it may be possible to obtain these values analytically if the graphs are small, or it may be necessary to obtain them by simulation or some other means.

5. Extension to General Networks

In network design it is common practice to reinforce the connections among a key set of central nodes, especially in the case where all communication must take place through these nodes. An example of the simplest such configuration of this type where the central nodes are connected in a cycle is shown in Figure 2.4.

The algorithms we have considered can be easily extended to handle such networks. Note first that without any modification to the algorithms, the network shown in Figure 2.4 can be reduced to a comparatively simple network consisting of the central nodes only. We would consider each central node as the root of a separate tree and analyze the tree using the algorithms of Section 3. When the algorithm terminates, the state vector at that node would reflect the structure of the entire tree rooted at the node. Analysis could then be carried

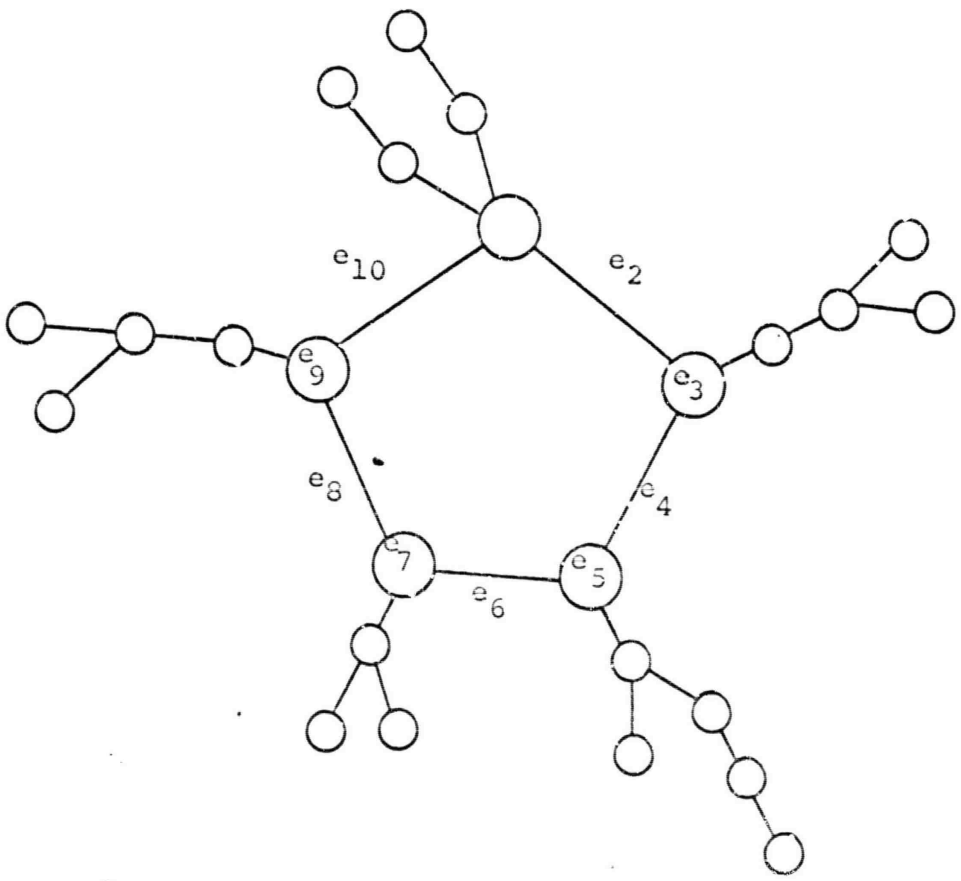


FIGURE 2.4

out, either analytically, or by simulation on the simplified network of central nodes.

If the simplified network is a loop, we can use the algorithms of Section 3 to analyze it by making the following observation: If any component in a loop fails the resulting network is a chain. A chain is a special kind of tree and can be analyzed using the recursive method.

Suppose we are given a cycle C_N, containing N elements (nodes and links) with ordering on the elements so that they are numbered $e_1, e_2, \ldots, e_N$ in a clockwise direction starting from some element, and that we desire to evaluate a reliability criterion, $RL(C_N)=RL_N$ on C_N. Consider RL_N^1 and RL_N^2 where:

$RL_N^1=RL_N$ given e_N is operative and

$RL_N^2=RL_N$ given e_N is failed.

Therefore $RL_N=RL_N^1QE(N)+RL_N^2PE(N)$ where $QE(N)$ is the probability that the element, e_N, works and $PE(N)$ is the probability it fails. RL_N^2 is easily evaluated by previous methods as the resulting network is a chain. To evaluate RL_N^1 consider RL_{N-1}^1 and RL_{N-1}^2 where:

$RL_{N-1}^1=RL_N^1$ given e_{N-1} is operative and

$RL_{N-1}^2=RL_N^1$ given e_{N-1} is failed.

Therefore $RL_N^1 = RL_{N-1}^1 QE(N-1) + RL_{N-1}^2 PE(N-1)$. This procedure can be repeated to yield a sequence RI_I^1 and RL_I^2 which are defined on disjoint segments of the total probability space and can, therefore be summed to yield the desired value, RL_N. All of these values, with the exception of RL_1 can be evaluated in terms of chains and can therefore be evaluated as before. RL_1 is defined on the cycle with all components operative, and is therefore easily evaluated. For example, if the cycle is composed of N nodes with weights, W(I), and if RL is the expected number of node pairs communicating then RL_1^1 is simply $\sum_{I=1}^{N-1} \sum_{J=I+1}^{N} W(I)W(J)$. Note also that the calculations of the RL_I^2 can be simplified by the observation that two adjacent operating elements e_i and e_{i+1} can be replaced by an equivalent element e* with:

$$W(*) = QE(I)W(I) + QE(I+1)W(I+1) \text{ and}$$

$$QE(*) = QE(I)QE(I+1).$$

This procedure replaces the evaluation of RL on a cycle with N evaluations of RL on chains. The order of computation is thus increased by a factor of N. The results can be extended still further to networks containing more than one cycle, but the order of computation will be increased in general by a factor of N_C (the number of elements in the cycle) for each cycle in

the network and will become excessive unless N or the number of cycles is small.

The same procedure is effective in analyzing networks of the form shown in Figure 2.5. It can be used first on each of the outer loops to obtain the expected number of node pairs communicating within the given loop, l_j, and the expected number of nodes in the loop which can communicate with $e_{i,1}$. These can then be used as initial conditions for S(I) and T(I) in the analysis of the central loop. If there are n nodes in the inner loop and K nodes in each of the outer loops, the entire procedure can be carried out in K^2n+n^2 steps.

6. Point Evaluation Versus Functional Evaluation

The calculations in the algorithms can be carried out in two ways. In the first way link and node probabilities, PL(I), QL(I), PN(I), QN(I). can be considered as numbers and the reliability criterion can be evaluated as a number. The evaluation can also be functional; that is, the reliability of the subtrees can be represented as polynomial functions of the link and node probabilities. This approach will of course require much more storage. The storage requirements are considerably reduced if all the node probabilities have the same value PN=1-QN and all the link probabilities have the same value PL=1-QL. In

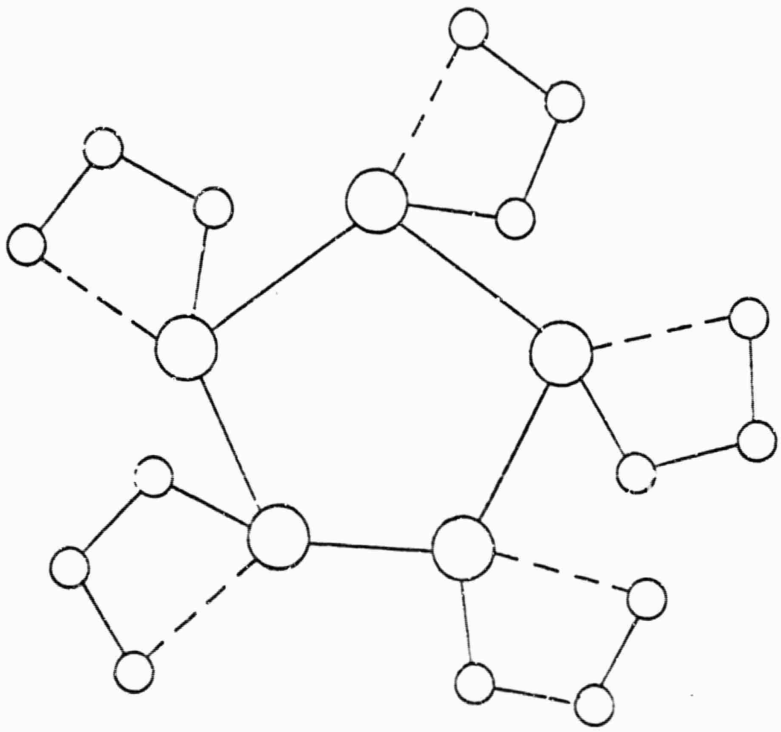


FIGURE 2.5

this case the various state components can be represented by the coefficients of power series in QN and QL.

As an example we carry out the calculations for the network in Figure 2.6 using as our criterion the expected number of node pairs communicating. We assume all links are operative with probability QL=p and all nodes operative with probability QN=q.

Initialization: $S(I)=q$, $T(I)=0$, $I=1, 2, 3, 4, 5, 6$.

I=6: $J=F(6)=3$

$T(3):=T(3)+T(6)+S(6)S(3)p$

$=0+0+qqp$

$=q^2p$

$S(3):=S(3)+S(6)qp$

$=q+qqp$

$=q+q^2p$

I=5: $J=F(5)=3$

$T(3):=T(3)+T(5)+S(5)S(3)p$

$=q^2p+0+q(q+q^2p)p$

$=2q^2p+q^3p^2$

$S(3):=S(3)+S(5)qp$

$=q+q^2p+qqp$

$=q+2q^2p$

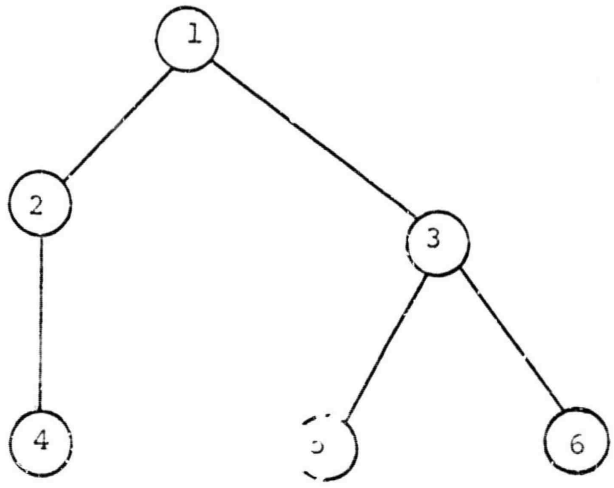


FIGURE 2.6

I=4: J=F(4)=2

$$T(2) := T(2)+T(4)+S(4)p$$
$$= q^2p$$
$$S(2) := S(2)+S(4)qp$$
$$= q+q^2p$$

I=3: J=F(3)=1

$$T(1) := T(1)+T(3)+S(3)S(1)p$$
$$= (2q^2p+q^3p^2)+(q+2q^2p)qp$$
$$= 3q^2p+3q^3p^2$$
$$S(1) := S(1)+S(3)qp$$
$$= q+(q+2q^2p)qp$$

I=2: J=F(2)=1

$$T(1) := T(1)+T(2)+S(2)S(1)p$$
$$= (3q^2p+3q^3p^2)+q^2p+(q+q^2p)(q+q^2p+2q^3p^2)p$$
$$= 5q^2p+5q^3p^2+3q^4p^3+2q^5p^4$$
$$S(1) := S(1)+S(3)qp$$
$$= (q+q^2p+2q^3p^2)+(q+q^2p)qp$$
$$= q+2q^2p+3q^3p^2$$

Note that the highest order polynomial, q^5p^4, corresponds to the longest path between two nodes (4 and 5). Note also that all terms in S(1) and T(1) are of the form $q^{i+1}p^i$. Thus, we could have simplified the calculations by considering

an equivalent tree with invulnerable links and nodes with probability of operation r=pq, except for the root, which still has probability of operation q.

III. A NEW ALGORITHM FOR MINIMUM SPANNING TREE CALCULATION

1. Introduction and Summary

A minimum spanning tree (also known as a shortest tree) is a tree in a networks whose total sum of link lengths (or costs) is as small as possible. Finding a minimum spanning tree is one of the most common and most important calculations in network analysis. Minimum spanning trees have been shown to be useful in reliability analysis (a new application), least cost electrical wiring, minimum cost connecting communication and transportation networks, minimum stress networks, clustering and numerical taxonomy, travelling salesman problems, multiterminal network flows, and Telpak routing.

Currently the most favored algorithm for finding minimum spanning trees is one due to Prim [1957] and Dijkstra [1959] . This algorithm takes on the order of n^2 computations where n is the number of nodes. It is simple to code, conservative of computer storage and is the fastest known method for complete networks. However, it has the unfortunate characteristic that the number of operations is not substantially reduced when the network is sparse, that is when the ratio of links to nodes is small as is the case in most practical networks such as ARPANET. Moreover, it is

too inflexible for use in network reliability applications. This led NAC to re-examine an earlier solution approach due to Krushal. By judicious use of list processing techniques and modern sorting techniques, the computation for this method became of the order m log m where m is the number of links. For complete networks $m = n(n-1)/2$ and Prim's algorithm is faster. However, in many applications in particular in the reliability analysis of the ARPA network $m \sim 2n$ in which case NAC's version of Kruskal's Algorithm [3rd and 4th Semi--annual reports] is much more efficient. Moreover, NAC's version of Kruskal's algorithm is much more flexible although at the cost of increased complexity of the algorithm.

Here we report on a dramatic improvement of Kruskal's algorithm which makes it competative with Prim's for complete networks also. Thus NAC's version of Prim's algorithm is competitive in computation time for nearly complete networks, is much superior for sparse networks and is much more flexible. The only remaining advantage for Prim's algorithm is that in certain situations, the storage requirements for nearly complete networks is less for Prim's algorithm than for the Krushal algorithm.

To be more specific we consider a network with nodes $N = \{1,\ldots,n\}$ and a set of m links A. Furthermore, each

link, $(i,j) \in A$ going from i to j, has a length λ_{ij} associated with it. We then ask what is the spanning tree of shortest total length for N. A generalization we will also consider is to find the shortest spanning forest with a fixed number, k, of components.

These very simple problems in graph theory have many practical applications. The most obvious application of minimum length spanning trees (MSTs) is to minimum connecting networks. Thus, if one wants to connect n points using the shortest network the solution is a MST (assuming there is no cycle or links with negative length). This fact has been used in transportation problems, communication design problems, and problems of wiring points together using minimum wiring in electronic wiring problems [Loberman and Weinberger: 1957]. Kalaba [1964] considered the following type of reliability problem on a network. Suppose with each link (i,j) there is associated a stress s_{ij}. The problem is to find a minimum stress path connecting the two given nodes; that is, a path connecting the two nodes such that the maximum stress for a link on the chain is minimized over all chains connecting the two given nodes. It turns out that the path between two nodes determined by a MST is a minimum stress path.

An application in which minimum spanning forests are of interest is in clustering analysis under the name of single linkage cluster analysis [Gower and Ross: 1969] [Zahn: 1971]. Suppose we have a set of points, S, and a function $\rho(i,j)$ which is a measure of the similarity of the points i and j. A family of subsets $\{C_m\}$ of S form a δ family of clusters if for each cluster C_m and each pair of nodes i and j in C_m, there is a sequence $i=i_1,\ldots, i_K=j$ with $\rho(i_k,i_{k+1}) \leq \delta$ for k=1,..., K-1 and for every pair of nodes i and j in different clusters $\rho(i,j) > \delta$. For a given δ the δ family is unique and corresponds to the components of a minimal spanning forest over all spanning forests with the same number of components.

A final application which motivated our interest is Monte Carlo simulation of network reliability. Suppose we have a network in which the links have a probability p of failing or q=1-p of not failing. We wish to investigate the probability of the network "failing." The network "fails" if it becomes disconnected. In any but the simplest cases, exact analysis is prohibitively difficult. Monte Carlo simulation then becomes attractive [Van Slyke and Frank: 1972]. The straightforward approach is to generate a random number $r_{i,j}$ for each link (i,j); if the random number $r_{i,j}$ is greater than q the link is removed; otherwise it stays in.

The resulting subnetwork is then examined to see if it is connected. The procedure is then repeated and an estimate is generated in the obvious way.

However, in most practical situations the probability of the network being disconnected is desired for a range of values for q. Suppose we want to find the probability the network is disconnected, h(q), for all q between 0 and 1 by Monte Carlo simulation. A possible method is to take the link with the smallest $r_{i,j}$, then the link with the next smallest $r_{i,j}$, and so on, until a connected graph is obtained. Let the last link have $r_{i,j} = q^0$. Then for $q < q^0$ the net is disconnected for this one sample and for $q \geq q^0$ the network is connected. Thus we get one sample for every value of q. We then generate a new set of random numbers for the links and obtain a second sample for each value of q and continue until the variance of the estimate is sufficiently small. We then use the fraction of the times the network was not connected as an estimate for h(q). It turns out that q^0 can be efficiently determined by finding a minimum spanning tree using $r_{i,j}$ as the length of link (i,j).

MST's have also been applied to multiterminal network flow analysis [Gomory and Hu: 1961] and to the solution

of traveling salesman problems [Held and Karp: 1970], [Held and Karp: 1971].

2. A History of Minimum Spanning Tree Calculations

The first major contribution to the theory of MST's was by Kruskal [Kruskal: 1956] although Choquet in 1938 and, according to Kruskal, Boruvka in 1926 did some some earlier work. Kruskal's major contribution was to show that a "greedy" algorithm [Edmonds: 1971] could be used to find minimum spanning trees. Specifically, he showed that an MST may be obtained by repeating the following step:

> Take the shortest link which has not been chosen or discarded. If it does not form a cycle with some of the previously chosen links, add it to the chosen links; otherwise discard it.

This algorithm is deceptively simple. The means of implementation on a computer is not obvious. The first attempts [Obruca: 1964] were of the following form: given an nxn matrix $(l_{i,j})$ of link lengths, find the smallest entry. Do a labeling procedure to find if the link forms

a cycle with the previously chosen link; if it does not, save the link, otherwise discard it. In either case, make the length of the link plus infinity and repeat the process. Searching the matrix and examining for loops involves on the order of n^2 comparisons which must be done on the order of n times so the running time for the algorithm in this form is cubic in n.

Shortly thereafter Prim [1957] and Dijkstra [1959] proposed an algorithm which takes on the order of n^2 operations. It is based on the following theorem: a tree is an MST if and only if for every $S \subset N$ there is in the tree a link of shortest length among all those connecting a node in S to a node in N-S [Rosenstiehl: 1967]. At every step of the algorithm we have a subset of the nodes S and a minimum spanning tree on S. We then find a shortest link (i,j) with $i \in S$ and $j \in N-S$ and add j to S and repeat. Slightly more formally the algorithm is:

Prim's Algorithm:

Step 0 (Initialization): Set $S_1 = \{1\}$, $k=1$, $d_1=0$. Set $d_j = c(1,j)$ if $(1,j) \in A$, $d_j=\infty$ otherwise. Set $f_1=0$, $f_j=1$ if $(1,j) \in A$, and $f_j=0$ otherwise. Go to Step 1.

Step 1 (Enlarge S_k by one node): Let $d_{j^*}=\text{Min}\{d_i: i \in N-S_k\}$.

If $d_{j*} = \infty$ go to Step 3; otherwise set $S_{k+1} = S_k \cup \{j*\}$ and go to Step 2.

Step 2 (Update distances across cut): For $j \in N-S_{k+1}$ set $d_j = \text{Min}\{d_j, \ell_{j*,j}\}$ and set $f_j = j*$ if $d_j = \ell_{j*,j}$. Set k=k+1. If k=n stop. Otherwise go to Step 1.

Step 3 (Network not connected, start new component): Let j be any node in $N-S_k$. Set $f_j = 0$ and go to Step 2.

The total number of operations in both Step 1 and Step 2 are quadratic in n. Moreover, since the number of links in a complete graph, n(n-1)/2, is also quadratic in n and since in general all links must be examined, the order of computation cannot be reduced to a lower order than quadratic for complete graphs. Unfortunately, even if the graph is not complete, the order of calculation is still quadratic since the minimization in Step 1 cannot be simplified in an easy way to take advantage of a network which is sparse, i.e., with m/n small.

The next stage of development was to realize that if the link lengths in Kruskal's Algorithm were presorted, efficient sort algorithms could be utilized. Conceptually then Kruskal's Algorithm would take place in two passes. First, the links are

sorted with respect to length. This takes on the order of $m\log_2 m$ operations. Then the links are introduced in order of length until a spanning tree is obtained. It turns out that the order of computation of the second pass is dominated by the computations involved in the first pass. Thus, if the networks are sparse and, say, the number of links, m, grows linearly with n rather than quadratically, Kruskal's Algorithm becomes faster than Prim's; on the other hand, for complete graphs $m\log_2 m$ looks like $\frac{n(n-1)}{2} \log_2 [\frac{n(n-1)}{2}]$ which grows faster than the order of computation n^2 required in Prim's Algorithm.

Treesort [Floyd:1962][Floyd:1964][Williams:1964] is particularly useful for use with Kruskal's Algorithm. An informal description of Treesort along with some of its properties are given in Appendix A. In the next section we turn to a problem even simpler than the MST problem: namely, that of finding out whether a network is connected or not. The solution to this problem furnishes an efficient procedure for the second pass in the improved Kruskal Algorithm.

3. Finding Components of a Graph and Spanning Forests

Finding out whether an undirected graph is connected or not in an efficient manner is not without interest in itself. In the Monte Carlo simulation of network reliability for a single

value of q, for example, determining the connectivity of a graph must be carried out thousands of times so it is worthwhile to find fast algorithms. Given the graph in node adjacency form, a very efficient method of determining the components is the following:

Algorithm A:

Step 0 (Initialization): Set i=1, j=1, S=Ø. Label node i=1 with component label j=1.

Step 1 (Look at new link): Find the next node i' adjacent to i; if there are none, go to Step 3. If node i' is not alrea in a component, go to Step 2. If node i' is already labeled with a component number, repeat Step 1.

Step 2 (Add a node to current component): Label the node i' with the current component label j and add the index of the labeled node to the stack S. Return to Step 1.

Step 3 (Scan a new node): Remove a node index i'' from S and set i equal to i". Go to Step 1. If S is empty go to Step 4.

Step 4 (Current component complete--start a new one): Set k to k+1. If k>1, we're done; otherwise, if node k is unlabeled, set i to equal k and set j to j+1. Go to Step 1. If node k is labeled, repeat Step 4.

This algorithm terminates with each component having a different label. If the links (i,i') occurring in Step 2 are saved, one also obtains a spanning forest. The order of computation is linear in n and m although if the graph is nearly complete, the number of links m is quadratic in n. If one is only interested in determining if the graph is connected or not, the algorithm can be terminated the first time Step 4 is encountered. This algorithm is probably close to being optimally efficient if the links are given in node adjacency form. Ushakov[1967] has proposed a similar algorithm which makes extensive use of logical operators on vectors which for many computers would allow savings in storage and computation time. However, he uses a node adjacency representation in matrix form which requires n^2 storage locations which may largely be wasted if the graph is sparse. Moreover, for each node he has to search an n-vector to find the first non-zero element. This could lead to a number of operations on the order of n^2 if there is no special machine instruction for rapidly carrying out this operation. Logically, the two algorithms are equivalent.

Algorithm A also has the disadvantage that the links incident to a node must all be scanned before links incident to other nodes can be worked on. This is necessary in order to

avoid relabeling nodes. For example, this restriction prevents one from adding in a simple way links to a graph already analyzed. A slightly slower but much more flexible algorithm [Van Slyke, and H.Frank:1972] is:

Algorithm B:

Step 0 (Initialization): Start with $A_0=\emptyset$ and assign each node a separate component label. Set k=0 and go to Step 1.

Step 1 (new link): Add a link $a_k=(i_k,j_k)$ to A_k to form A_{k+1} (if there are no remaining links; i.e., A_k=A stop). Examine the component labels of i_k and j_k; if they are the same, repeat Step 1 with k set to k+1. If not, go to Step 2.

Step 2 (Join components): Change all the node labels which are the same as the label of i_k (including i_k's label) to the label of j_k. Set k to k+1 and go to Step 1.

The order of computation is dominated by the relabeling in Step 2 which occurs n-c times where c is the number of components. Using a straightforward implementation [Berge: 1962] [Berge, Ghouila - Houri:1965] [Seppanen:1970] each time through Step 2, the labels on all n nodes have to be checked in order to relabel. Thus, on the order of n^2 operations are involved with relabeling.

In the version of the algorithm used by Van Slyke and Frank [1972] a list structure was maintained so that only nodes for

which the labels are changed are considered. Further, the number of nodes in each component was maintained so that it was possible to change the labels on the smaller of the two components joined in Step 2. This reduces the maximum order of computation to $n\log_2 n$ plus a term linear in m. This increase in speed by using list structures does incur an expense in storage requirements. Knuth[1968] and Read[1969] have proposed maintaining component membership using tree data structures rather than the explicit relabeling used in Step 2 of Algorithm B. However, in this approach determining whether a candidate link connects two nodes in the same or in different components takes several steps compared to the one comparison required by Algorithm B and is therefore less efficient.

4. New Developments in MST Calculation

Until recently, the most efficient methods for calculating minimum spanning trees or forests was to use Prim's algorithm for nearly complete graphs which involves on the order of n^2 calculations or using the Kruskal Algorithm with Treesort and Algorithm B. The sorting pass takes on the order of $m\log_2 m$ calculations while the second pass involves $n\log_2 n$ dependence on the number of nodes n and depends linearly on the number of links. Thus, for sparse graphs where $m\log_2 m$ is small compared to n^2,

the modified Kruskal algorithm will be faster. Important considerations other than speed of computation will be discussed in Section 6; here we report on efforts to develop MST algorithms which are uniformly fast over the full range of sparseness.

The first approach is to notice that the main expense in the modified Kruskal Algorithm is in the sorting which takes in general $m\log_2 m$ operations and to notice that most of the links are not considered because they make cycles with shorter links. In Treesort (Appendix A) applied to the list of link lengths, the list is first arranged into a binary tree which is a "heap"; that is, each link length is no longer than its descendants in the tree. This takes about m interchanges and 2m comparisons at the worst. Then the top link corresponding to the top of the heap is considered via Step 1 of Algorithm B for the MST. Then the link length is deleted from the heap and a new link length corresponding to link (i,j), say, is taken from the bottom to the top and the heap restored by a sift-up. The sift-up takes at most $2\log_2 m-1$ interchanges and at most $2\log_2 m-2$ comparisons to restore the heap. Often the sift-up can be saved by comparing the component labels of i and j in Algorithm B. If they are the same, the link forms a cycle with shorter link and can be discarded immediately. Using this approach, the sorting cost is

on the order of $2m+k \log_2 m$ where k is the number of links examined in Algorithm B before a spanning tree is obtained since only the links actually considered for the MST are sorted. In general, one may have to examine all m branches but for nearly complete graphs this is unlikely. Experimental verification of this is found in Section 5. There it is shown that using this further modification to Kruskal's Algorithm, it becomes nearly as efficient as Prim's Algorithm for complete graphs and is still much better than Prim's Algorithm for sparse graphs.

However, Prim's method can be improved also. Here we follow the approach due to E. Johnson [1972] who applied the idea to Dijkstra's shortest path algorithm. We use Treesort to determine $d_j = \text{Min } d_i$ in Step 1 of Prim's Algorithm. We assume the d_i form a "heap" (see Appendix A). The top of the heap is d_{j*} which is then removed from the heap. Then, in Step 2 of Prim's Algorithm some of the d_j become smaller and are modified. Next, a d_j from the bottom of the heap is moved to the top. Finally, the heap is restored. At the worst, each restoration of the heap takes a number of operations linear in n and usually considerably less is needed especially if the network is sparse. Even if it is not sparse, many of the d_j do not change; furthermore, all but one of the d_j which change decrease in value so that the

"sift-up procedure" of Treesort takes on a particularly simple form (See Appendix A). In the next section we present the results of numerical experiments on these algorithms.

5. Numerical Experiments

Numerical experiments were carried out on randomly generated networks. For given n and m random graphs of m links and n nodes were generated. Then, a random length between 0 and 1 were generated for each link. The distribution of lengths is of no importance since as Rosenstiehl [1976] pointed out, any equivalent pre-ordering would give the same results; thus, any method which generates random permutations of the links will suffice. Three series, four networks each, were used. These are given in Table 3.1.

Series		(n,m)
1	m=n-1	(10,9) (50,49) (100,99) (500,499)
2	m=3n	(10,30) (50,150) (100,300) (200,600)
3	m=n(n-1)/2	(5,10) (10,45) (20,190) (40,780)

Table 3.1. Network Series Used

One hundred samples of each of the 12 network sizes were analyzed by each of four algorithms: Prim's, Modified Kruskal, Prim's with sorting, Modified Kruskal's with partial sorting.

Each algorithm was presented with exactly the same networks. For each of the algorithms and each of the network sizes, the analysis time for each of a hundred trials was obtained. The maximum over 100 trials, the average over the 100 trials and the standard deviation over the hundred trials was recorded. The computer clock gives results in milliseconds and the clock routine itself takes less than one half millisecond. The results are presented in tabular form in Table 3.2 and in graphical form in Figure 3.1. As is suggested by theory, Prim's works best for the complete graphs and Kruskal's works better for sparse graphs. The two algorithms using sophisticated versions of Trees it yield good results over a wider range of sparsity. The modified Kruskal Algorithm with partial sorting is apparently the best if speed over a wide range of sparseness is the criterion.

Table 3.2. Comparison of Running Time Average over 100 Trials for Four MST Algorithms

n	m	Kruskal	Kruskal/ Partial Sort	Prim	Prim Sort
5	10	2.2	1.7	1.59	2.24
10	45	10.2	5.94	4.64	6.42
20	190	51.6	19.6	16.07	20.13
40	780	263.65	67.79	58.48	68.24
10	9	2.29	2.34	3.36	3.91
50	49	14.62	14.46	16.70	30.66
100	99	32.28	31.81	106.75	74.27
500	499	193.98	195.35	*	536.76
10	30	6.71	4.71	4.37	5.91
50	150	42.44	35.97	59.03	[illegible]
100	300	93.41	83.30	1[illegible].99	[illegible]
200	600	206.22	199.49	8[illegible].[illegible]	[illegible]

*(n=500, m=499 not computed for Prim.
In milliseconds on a CDC 6600 Computer.

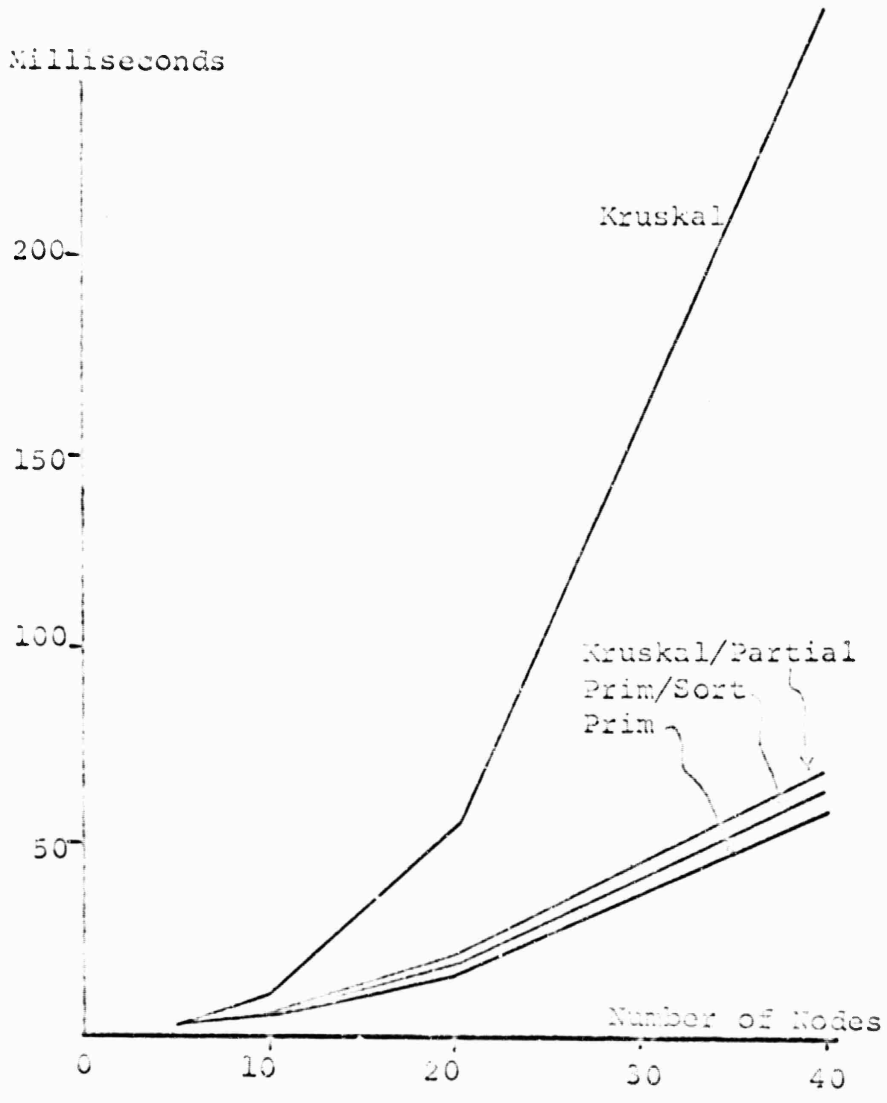


FIGURE 3.1(a)

TIME IN MILLISECONDS TO FIND MST
m=n(n-1)/2

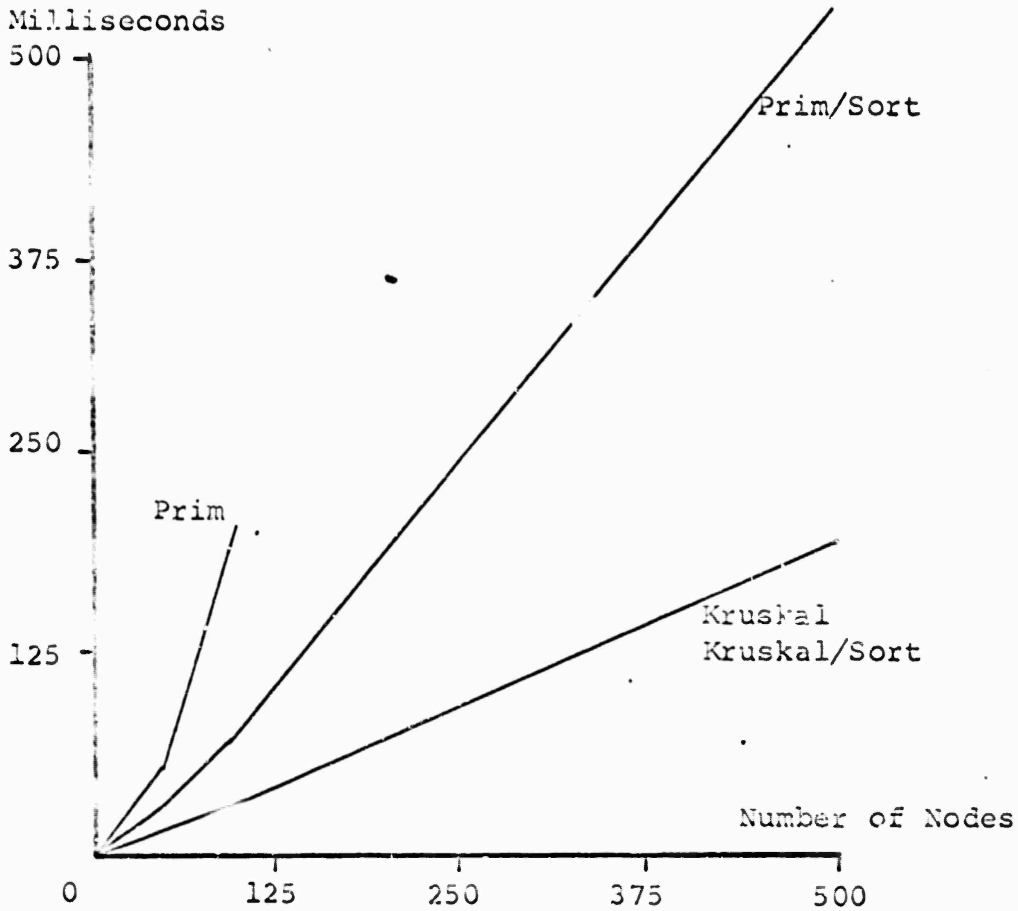


FIGURE 3.1(b)

TIME IN MILLISECONDS TO FIND MST
m=n-1

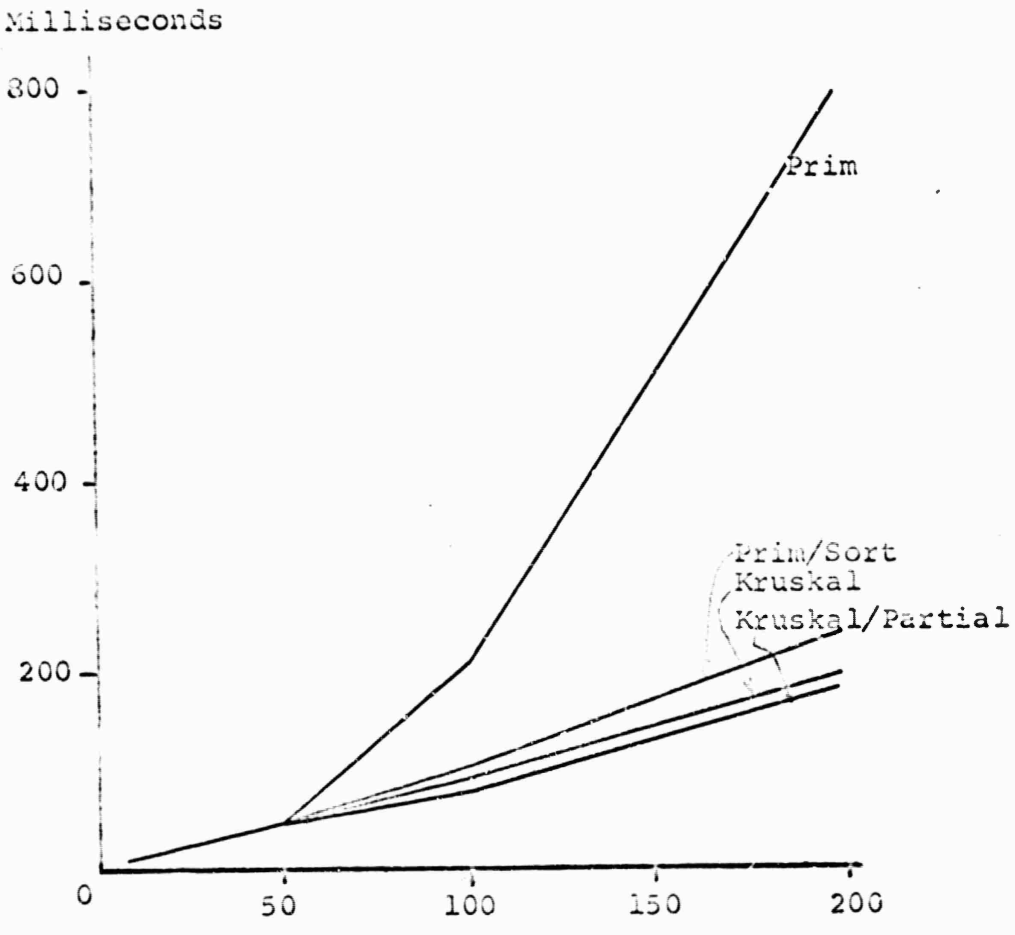


FIGURE 3.1(c)

TIME IN MILLISECONDS TO FIND MST
m=3n

67.

6. Summary and Conclusion

Speed is not the only measure in choosing an algorithm for MST calculations. Other important considerations are: storage requirements, form of input, availability of algorithm and difficulty of implementation, the particular application, and the number of problems to be solved.

Prim's algorithm is superior in many of these respects. It is very fast for large nearly complete networks; the algorithm is easy to implement; the storage requirements are quite small especially if the network is complete and the link lengths are a simple function of the end nodes, say Euclidean distance. Then the link lengths need not be stored at all but are generated once as needed in Step 2. With the addition of Treesort in Step 1 of the algorithm, Prim's Algorithm becomes much more useful for sparse graphs; however, the algorithm becomes considerably more complex and some speed is lost in analyzing complete graphs.

Prim's Algorithm also has other disadvantages. It requires the link information to be presented in node incidence format and it cannot be used for determining minimum spanning forests with various numbers of components. This latter problem makes Prim's Algorithm somewhat unsuitable for single linkage cluster analysis, reliability analysis of networks, and multiterminal network flow analysis.

The modified version of Kruskal's Algorithm is very good for determining MST or minimum spanning forests on sparse networks and it accepts the links in any form. When it is used with partial sorting, it gives the best results over the complete range of sparsity. Moreover, changing to partial sorting can be done very easily. Both versions of Kruskal's Algorithm require a relatively large amount of storage because of the list structures required by Algorithm B and in all cases require the storage of the complete list of link lengths.

We close by analyzing two specific applications. These are Monte Carlo network reliability analysis and single linkage cluster analysis. Most practical pipelin., transportation, or communication networks are sparse and of reasonable size. Moreover, for simluation of reliability many hundreds or thousands of trials must be computed. Finally, if expected fraction of node pairs communicating is used as a criterion of reliability [Van Slyke and Frank: 1972] rather than probability or being connected, it turns out that minimum spanning forests are required. Thus, the algorithm must be fast for sparse graphs, and it must be capable of determining minimum spanning forests. Moreover, storage is usually not a problem. Thus modified Kruskal is ideal for this application.

Single linkage cluster analysis presents a slightly more difficult case. Here in general the network is complete since every pair of points is related, which would seem to indicate Prim's Algorithm; however, the main results required are minimum spanning forests with various numbers of components (in order to get the ℓ clusters) which is not available from Prim's Algorithm. The best compromise until now was suggested by Gower and Ross [1969] which was to do Prim's Algorithm and toss out all links not in the MST. This leaves only n-1 links. Then a form of Kruskal's Algorithm is applied to find the minimum spanning forests with various numbers of components. This approach is still desirable when storage is a problem and link length is a simple function of the end nodes so that the link list need not be stored. If there is sufficient storage, the modified Kruskal Algorithm with partial sorting should be much faster.

APPENDIX A: Treesort

A list of $m=2^k-1$ numbers n_2, ..., n_{2^k-1} can be identified with a rooted binary tree with k levels where n_1 is at the root, n_2 and n_3 are at the next level and, in general n_{2i} and n_{2i+1} at level $\lambda+1$ are connected to n_i at level λ. In Figure 3.2 this mapping is illustrated for k=4.

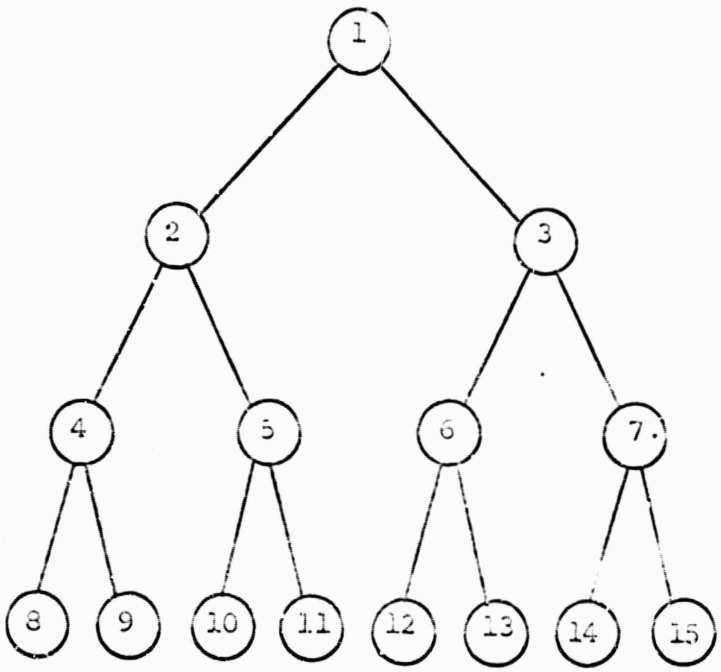


FIGURE 3.2

The list $L=(n_1, \ldots, n_{2^k-1})$, or equivalently the binary tree associated with it is called a heap if $n_i \leq n_{2i}$ and $n_i \leq n_{2i-1}$. The first observation is that if L is a heap then $n_1 \leq n_i$ for $i=1, \ldots, 2^k-1$. If for a given list $L=(n_1, \ldots, n_\ell)$ of length ℓ, $\ell \neq 2^k-1$ for some k, we choose the smallest k such that $2^k-1 \geq \ell$ and fill the unused slots with $+\infty$. The number n_i is called the father of n_{2i} and n_{2i+1} and n_{2i} and n_{2i+1} are called the sons of n_i. An element n_i determines a unique subtree consisting of its sons, its son's sons, and so on. In Figure 2, n_3 determines the trees made up of n_3, n_6, n_7, n_{12}, n_{13}, n_{14}, and n_{15}. A fundamental operation in Treesort is taking an element n_i for which the subtrees determined by n_i's two sons n_{2i} and n_{2i+1} are heaps and by permuting elements forming a heap which is a subtree determined by n_i. This is called a sift-up of n_i, although the way we drew Figure 3.2 it should more properly be called a sift-down. An even more basic operation is an exchange-test at n_i. This is carried out in two steps. First, n_{2i} and n_{2i+1} are compared; then the smaller of the two is compared with n_i. If n_i is larger n_i is interchanged with the smaller and n_{2i} and n_{2i+1}. This involves two comparisons and (possibly) one interchange.

A sift-up of n_i at level ℓ is accomplished by performing an exchange-test at n_i. If there is an interchange, an exchange-test is performed at the new position of n_i which is necessarily at level $\ell+1$. The procedure is continued until there is no interchange or until level k is reached [London: 1970]. If the subtree determined by n_i has k levels then sift-up takes at most k-1 exchange-tests for a total of 2k-2 comparisons and k-1 interchanges.

The first part of Treesort consists of establishing a heap. This is done recursively using sift-up. The subtrees determined by the elements at level k-1 can be made into heaps in one exchange-test each. Then the elements at level k-2 are made into heaps using sift-ups. One works up the tree until finally the subtree determined by n_1, which is the entire tree, is made into a heap. An element a level ℓ determines a subtree with $(k-\ell+1)$ levels hence a sift-up could take $k-\ell$ interchanges and $2k-2\ell$ comparisons. Since there are $2^{\ell-1}$ elements at level in the worst case

$$\sum_{\ell=1}^{k-1} (k-\ell)\ 2^{\ell-1} = 2^k-k-1$$

interchanges and twice that many comparisons would be needed.

Since the length m of the list is 2^{k-1} we have that $k=\log_2 m$ and the order of calculation in the worst case is $m-\log_2 m$ interchanges and $2m-2\ \log_2 m$ comparisons.

Since we have a heap, the top element is the smallest element of the list. We now enter the second phase of Treesort. The top element is removed from the heap and saved. The last finite element of the tree is then put at the top and sift-up is carried out until it finds its proper level. The newest top element (the second smallest element of the original list) is removed and the last finite element is brought to the top. The number of interchanges in the worst case is λ-1 if the tree has λ levels remaining. In general, there are $2^{\lambda-1}$ sift-ups carried out on λ level trees. Thus there are approximately

$$\sum_{2}^{k} (\lambda-1)\, 2^{\lambda-1} = 2+(k-2)2^{k}$$

interchanges and twice that amount of comparisons in the worst case for the second pass. This is on the order of $m\log_2 m$. If only the smallest r elements are required rather than a complete sort then the computation in the second part is of the order $r\log_2 m$.

Treesort can also be carried out in place. That is, the elements removed from the top of the heap can be stored in reverse order in the unused bottom of the heap.

In Figure 3.3 Treesort is used to find the two smallest elements of the list {6, 14, 11, 8, 17, 20, 2}.

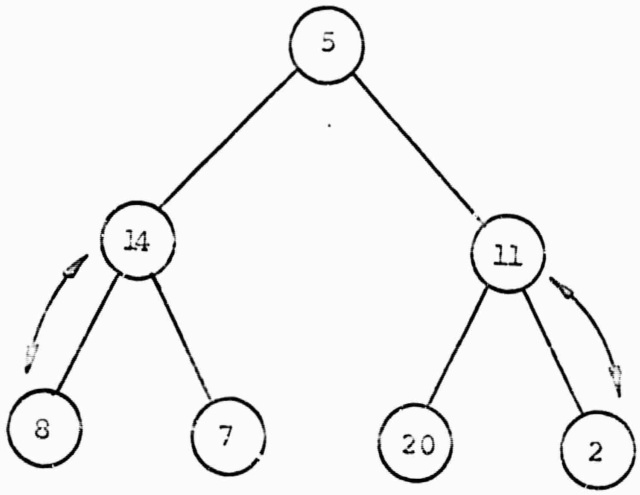


FIGURE 3.3(a)

LEVEL 2 INTERCHANGE TESTS

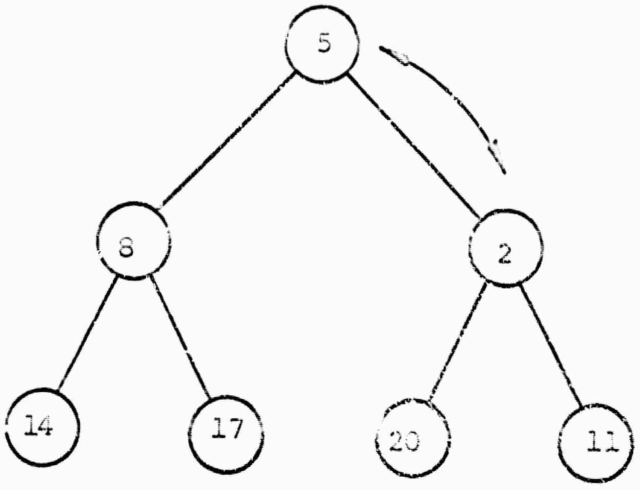


FIGURE 3.3(b)

LEVEL 1 EXCHANGE TESTS

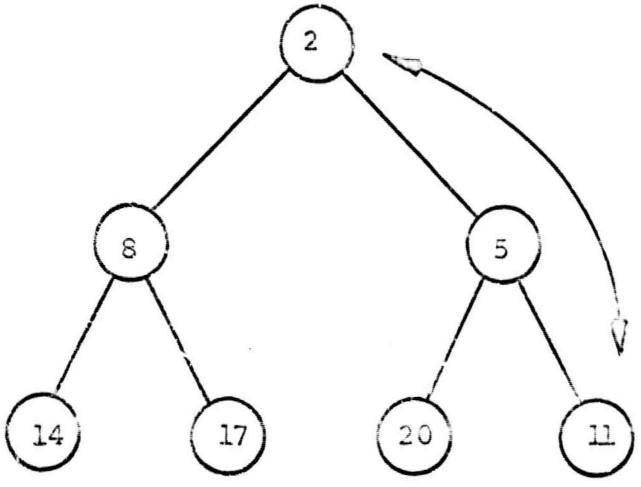


FIGURE 3.3(c)

HEAP

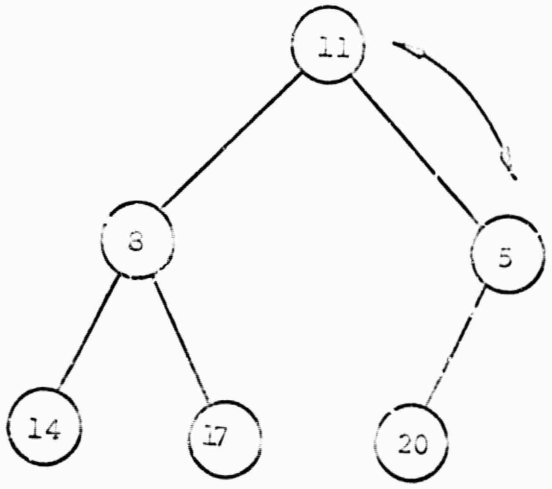


FIGURE 3.3(d)

SMALLEST ELEMENT FOUND

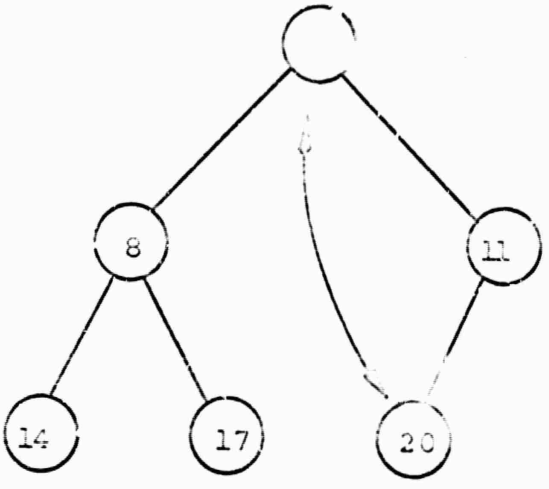


FIGURE 3.3(e)

HEAP RESTORED

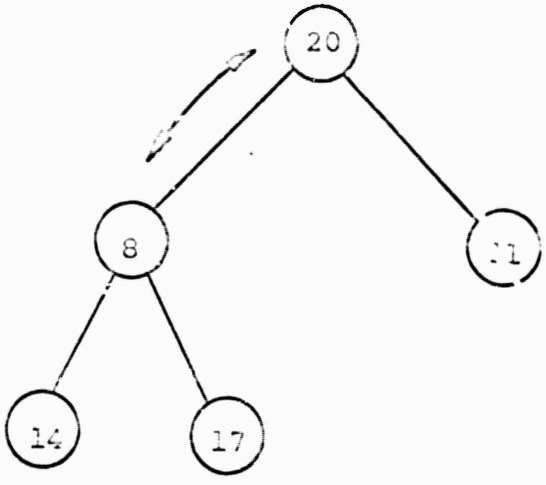


FIGURE 3.3(f)

TWO SMALLEST ELEMENTS OBTAINED

As a final aspect of Treesort we examine the question of re-establishing a heap when some of its elements are changed. As in establishing an initial heap in Treesort the remaking of a heap will involve sift-ups but not as many. The elements are scanned starting from the next to the bottom level as in the first part of Treesort except that a sift-up is done on an element if and only if the element itself is bigger than it was in the heap or if one of its sons is smaller than it originally was. If only decreases in value were made, as in the case in one of the algorithms for MST's one can proceed by immediately marking the father of all elements decreased. Do a sift-up on the last (lowest in heap) element marked, n_i. If there are any interchanges, mark the father of n_i. Then erase the mark on n_i and go to the marked element lowest in the heap and continue. In Figure 3.4 this process is illustrated.

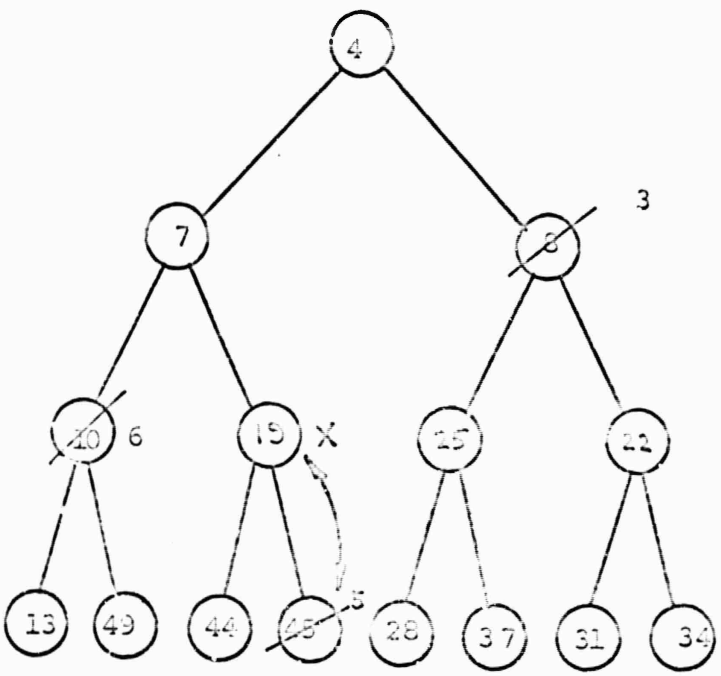


FIGURE 3.4(a)

32.

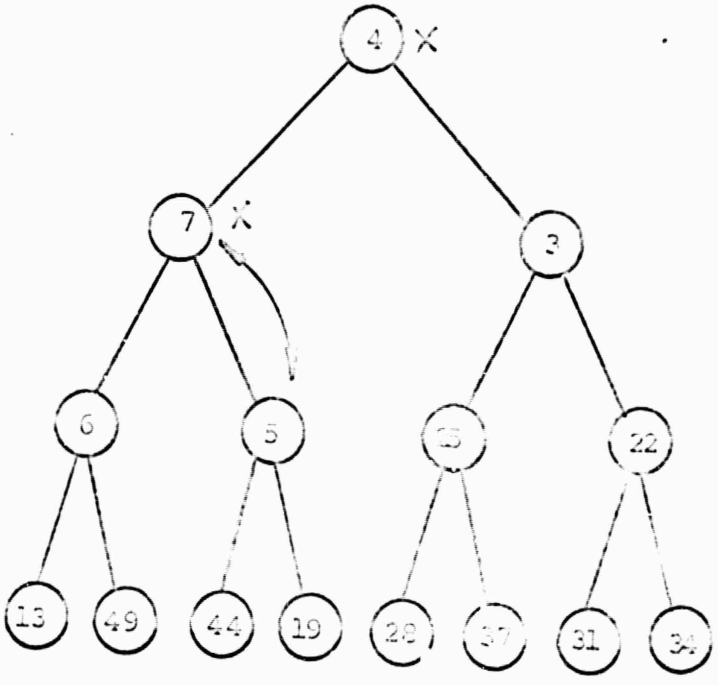


FIGURE 3.4 (b)

~~83.~~

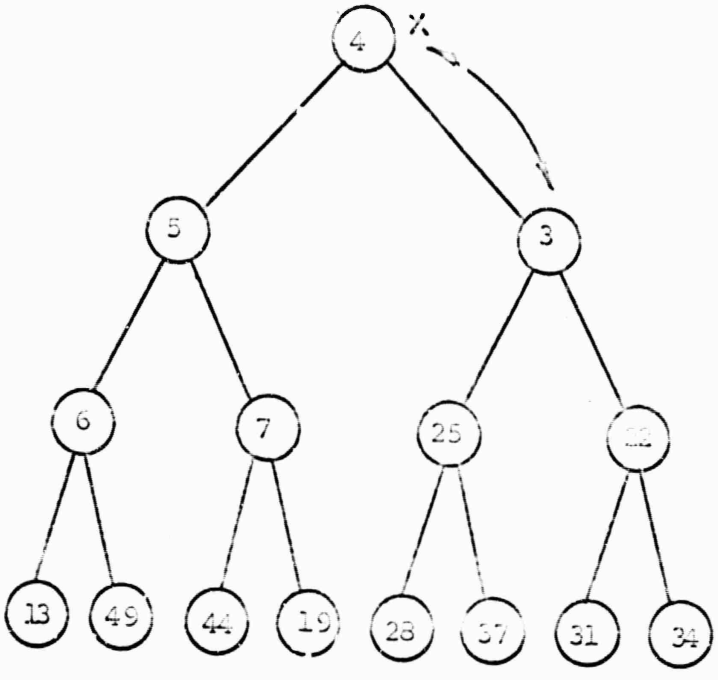


FIGURE 4(c)

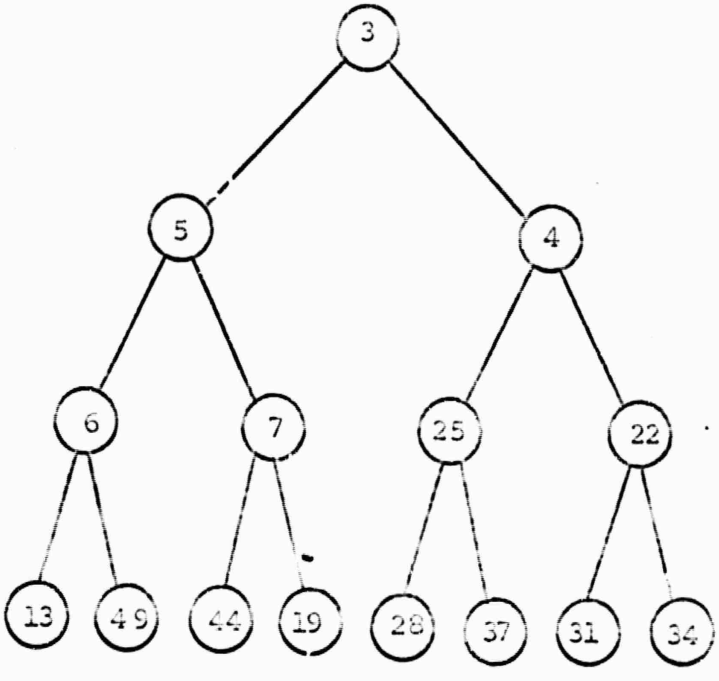


FIGURE 4(d)

es.

IV. REFERENCES

1. Loberman, H. and Weinberger, "Formal Procedures for Connecting Terminals With a Minimum Total Wire Lingth", JACM, 4, 428-437, 1957.

2. Kalaba, R., "Graph Theory and Automatic Control", in Applied Combinatorial Mathematics, E. Beckenbach (ed.) Wiley, 1964.

3. Gower, J.C. and Ross, G.J.S., "Minimum Spanning Trees and Single Linkage Cluster Analysis", Appl. Statistics, Vol. 18, No. 1 54-64, 1969.

4. Zahn, C.T. "Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters", IEEE Trans. on Computers, C-20, No. 1, January, 1971.

5. Van Slyke, R. and Frank, H., "Network Reliability Analysis I", Networks, Vol. 1, No. 3, 1972.

6. Gomory, R.E. and Hu, T.C., "Multi-Terminal Network Flows", J.SIAM, 9, No. 4, December, 1961.

7. Held, M. and Karp, R., "The Travelling Salesman Problem and Minimum Spanning Trees", Operations Research, 18, No.6, 1138-1162, Nov.-Dec. 1970.

8. Held, M. and Karp, R., "The Travelling Salesman Problem and Minimum Spanning Trees: II", Mathematical Programming, 1, No. 1, 1971.

9. Kruskal, B.,Jr. "On the Shortest Spanning Subtree of a graph and the Travelling Salesman Problem", Proc. Amer. Math. Soc., 7, 48-50, 1956.

10. Chouquet, G., "Etude de Certains Reseau de Route", C.R.Ac.Sc., 206, 310, 1938.

11. Borůvka, "On a Minimal Problem", Prace Moravske Pridovedecke Spodecnosti, 3, 1926.

12. Edmonds, J. "Matroids and the Greedy Algorithm", Mathematical Programming, 1, No.2, 1971.

13. Obruca, A., "Algo. 1 Mintree" Computer Bulletin, p. 67, Sept. 1964.

14. Prim, R.C., "Shortest Connection Networks and Some Generalizations," Bell System Tech. J., 1389-1401, November, 1957.

15. Dijkstra, E.W., "A Note on Two Problems in Connection with Graphs", Numerishe Mathematik, 1, 269-271, 1959.

16. Rosenstiehl, P., "L'Arbre Minimum d'un Graphe", in Theory of Graphs, P. Rosenstiehl, ed., 1967, Gordon and Breach, N.Y.

17. Floyd, R.,"Treesort", Algorithm 113, ACM Collected Algorithms, August 1962.

18. Floyd, R."Treesort 3", Algorithm 245, ACM Collected Algorithms, December, 1964.

19. Williams, J.W.J., "Heapsort", Algorithm 232, ACM Collected Algorithm, January, 1964.

20. Ushakov, I.A., "An Algorithm for Check' the Connectivity of a Graph", Izv. AU.Tckh.Kib., No. 4, 85-96, 1967, Trans: Eng. Cyb., No. 4, 82-83, 1967.

21. Berge, Theory of Graphs, 152-155, 1962, Wiley.

22. Berge, and Ghouila-Houri,A., Programming Games, and Transportation Networks, 177-182, 1965, Wiley.

23. Seppanen, "Spanning Tree (H)" Algorithm 399, Comm. ACM, 13, No. 10, October, 1970.

24. Knuth, P.E., The Art of Computer Programming: Volume--Fundamental Algorithms, Sections 2.3.3 and 2.3.4, 1968, Addison-Wesley.

25. Read, C., "Teaching Graph Theory to Computer", in Recent Progress in Combinatorics, W.T. Tutte (ed.) Academic, 1969.

26. Johnson, Personal Communication, 1972.

27. London, L., "Certification of Algorithm 245 (M1) Treesorts", Certification of Algorithm 245, ACM Collected Algorithms, 1970.